State of the System Year One

Luc Alexander Lüthi

December 27th, 2025

Contents

1 Programming Languages

I pivoted this year to researching in the programming language design space. The space is considered mature, but when I peered my eye at the past I was met with only questions. Tools rose and fell out of fashion and we dont entirely understand why, we just know that what we have now works better than what we had before, and it seems the best metric we have been able to come up with is that it simply feels better and easier to write successful efficient well structured programs.

1.1 The Current Meta

Structured imperative programming is at the forefront of the paradigm space, and for a good reason, its largely unparadigmatic. Most languages fall on a spectrum of how mathemetically based they are, many

language paradigms which fail are purists of some idea. Object oriented programmers have doctrine for their purity, functional programmers have proof for their purity, assembly has performance. But the truth of what will emerge as the dominant way to program will likely be a balance of all the best things in each programming world. That is where structured imperative programming lives. Its doesnt exist by virtue of strict rules about how we should express computing, it exists by being a combination of all which we have found to be the best way to express a small portion of the computing problem space, which gives stsrict rules about how the computation actually happens. This is something I didn't understand at the beginning of this year.

Functional programming had a brief rise in popularity between 2015 and 2025, this trend may continue but I can say with certainty that it will fall by the wayside, and the best portions of it will be scrapped and used for a new amalgamated form of structured imperative programming, like the Borg. The same will happen to logic programming whenever that makes it out of academia and onto the influencer sphere of programming. Each statement in a dominant imperative languages of the time can be functionally pure themselves, but a program as a whole in a real environment has global state, it has mutation. I care about real systems, and when I write my programs I care that they will be running on real computers. I want to write programs which pay attention to the the hardware they are running on and respect the environment they live on.

All purist programming paradigms will fall out of fashion because they will always miss one mark. Whether that mark be performance, debuggability, ease of expression, ease of readability, or a whole plethora of other checkboxes necessary for the success of a language as a tool for real work and real systems.

The current meta pays attention to everything only so much as it needs to. Modern languages are beginning to respect immutability default semantics, are having more interfaces and closures, but are doing away with what they dont need. This will continue to be the case, and so I believe it is in our best interest to study programming language design as far as there persist problems in the programming world. The largest problem I see is a matter of how to make programs bugfree and secure while retaining ease and depth of expression. That would be a huge win for the life of a programmer.

1.2 What Gives a Programming Language Power

When I think about what makes a programming language I think of a few simple metrics: what can it express, how easy is it to express things, are the easy ways to express things the correct ways to solve the problem, and what is its command over its environment. The last one seems unrelated to the rest but is very important, we write programs in the real world which will run on real machines. These machines have memory, each instruction takes a certain amount of time to execute, theres caching, this is all important for real computer programming. Almost no language confidently brings positive answers to all of my metrics of power.

1.3 What Makes a Programming Language Bad

There are many reasons why programming languages as a whole can be bad, but usually when people talk about why something in a language is bad, they talk about how it can be misused in a way which brings confusion, bugs, or inefficiency. The reality of a lot of these features is that they are very powerful features, and this power enabled misuse in the hands of a novice. A great example is C style macros. If used conservatively and in the hands of someone competent, they are an excellent way to get a small boost in ergonomics for a C program, but they are blanket considered bad practice in a lot of programming

spaces because if you encourage programmers to use them, novices wil misuse them to make a mess of their program.

This idea scales beyond simple features however, as languages which often argue themselves as the most "powerful" programming language ever created, are never popular choice anymore. This is not because they are lying, it is because unrestricted power is bad for a programming language for the same reason language features themselves are considered bad on a micro scale. Lisp can express anything, but has nothing holding it down, leading to the C macro problem but as a whole language.

1.4 What I Desire in a Programming Language

Realistically when we talk about power in languages in the real world, the metric changes for each language. Lis phas expressive power, C has computational power, Lean, to some degree, has constraint based power. It was at this point in my thinking that I made a note that it would be really nice to have the power of all three. What would that look like? A Lisp that has you reasoning about assembly or bytecode, constrained by some in language protocol or logical macro system for defining systems of constraint like type systems, borrow checkers, proof engines, etc. I mostly discarded this thought when I had it, writing a simple trivial prototype and then moving one, but at the end of the year that is what I ended up developing. If only I had taken the idea seriously when I initially had it.

The reason I started compiler programming and langauge programming as a whole is that I wanted to be able to express computation in my preferred way, while allowing myself a space where if I wanted to I could add or remove features myself to hyper currate my programming environment to what I believed was the perfect expressive form of a modern structured imperative Borg programming language. The first few attempts at this looked like the traditional static programming language, but evolved into what I now understand as a new different type of computing environment from a compiler, a meta compiler. These tools are not new but were largely abandoned around fifty years ago due to, largely, the systems they outputted being unconstrained soups of computation. After this half century it is clear that the direction of programming language design is headed toward constrained computation spaces. Rust is the first real industry ready pilot in this direction, and like many firsts, it kind of misses the mark, but its also the only real attempt at a non type theory based constriction of the computation space. There must be more constrictions out there which work, and work better than Rust.

1.5 Filling a Niche

To test a constraint system like a type system, or ar proof engine, or a borrow checker, or something completetly new like, for instance, a system which constrains the control flow of your program to what is achievable by a finite state automata rather than a Turing machine in order to rpeserve provability, you typically have to write an entire compiler/interpreter. This is arduous. My task switched in the middle of this year from being "how can I make my perfect programming language" to "how can I make an environment where I can express and iterate on forms of computational expression as effectively as possible so that I may find good constrictions of the language space".

By good I mean allows ease of expression, control over the computation environment, and constrains away as many bug cases as possible without encroaching on either of the other two metrics.

2 Forgotten Prototypes

There exists a number of simple virtual machines and failed compiler prototypes which precede any research and existed as learning exercises. These all took place during 2024. The only complete project from this time consisted of an 8 bit virtual machine. Programs in this language are very simple and not very capable.

NOP			
LDW	00 dst src	00000 ofs	
	01 dst src	2 byte ofs literal	
	10 dst	2 byte src address	
	11 dst	2 byte src literal	
LDB	00 dst src	00000 ofs	
	01 dst src	2 byte ofs literal	
	10 dst	2 byte src address	
STR	00 src dst	00000 ofs	
	01 src dst	2 byte ofs literal	
	10 src	2 byte dst address	
STB	00 src dst	00000 ofs	
	01 src dst	2 byte ofs literal	
	10 src	2 byte dst address	
LAR	00000 dst	4 bit src	
ADD	0 m dst op1	2 byte int or reg op2	
SUB	...	...	
MUL	...	...	
DIV	...	...	
MOD	...	...	
LSL	...	...	
LSR	...	...	
AND	...	...	
ORR	...	...	
XOR	...	...	
COM	0000 m reg	2 byte literal or reg	
PSH	0000 m reg	2 byte literal	
POP	00000 reg		
CMP	00000 op1	00000 op2	
JMP	metric byte	2 byte label address	
JSR	metric byte	2 byte label address	
RET			
INT	interrupt		

Comparison metrics

NC	No condition
EQ	Equal
NE	Not Equal

```
| LT | Less thanion  |
| GT | Greater than  |
| LE | Less Equal    |
| GE | Greater Equal |
‘--------------------’
```

```
Interrupts

.-------------------------------------------------------------------,
| OUT | Print ascii string to stdout | R0 <- address, R1 <- length  |
| KBD | Get keyboard key status      | (unimplemented)              |
| END | End the program and return   | R0 <- process return code    |
‘-------------------------------------------------------------------’
```

This technically comprises the first programming language I ever wrote, while tiny.

```
    LDW R0 #5
    LDW R1 #6
    PSH R0        ; push arg 1
    PSH R1        ; push arg 2
    JSR NC add    ; jump to subroutine
    POP R0        ; retrieve return value
    INT END

add:
    LAR R1 FP     ; load auxiliary register (frame pointer)
    ADD R1 R1 #x1 ; add 1 to get previous element in stack
    LDW R3 R1 #x8 ; load stack address offset 8 bytes from the frame pointer (arg 2)
    LDW R2 R1 #xc ; load stack address offset 12 bytes from the frame pointer (arg 1)
    ADD R2 R3 R2  ; function body
    PSH R2        ; push return value
    RET           ; return to previous stack frame

    LDW R0 #0       ; load program rom return code
    INT END         ; call interrupt to end program
```

Out of this era emerged a half done compiler project which really just needed a code generation pass, but I had little confidence in the quality of its code, being that it was my first attempt at a compiler. It was subsequently scrapped. I later restarted the project as Ink.

3 Protosrc

This project technically took place at the very end of 2024. It was tiny, was largely written as an exercise, and the entire language comprises of a simple bytecode, for which procedure calls can be abstracted away by nested S expression calls. Theres also a very tacked on macro system. The entire spec is available in the git README.

```
 ,---REGISTERS-------------------------------------.
 |  General  |  Special  |  Purpose                |
```

rA	IP	Instruction Pointer	
rB	SP	Stack Pointer	
rC	FP	Frame Pointer	
rD	SR	Status Register	<---.---BITS---
rE	CR	Call Argument Register	| 0 : Zero
rF	AR	Aux Register	| 1 : Carry
rG			| 2 : Sign
rH	^		| 3 : Over
rI	'--[These are preserved between calls		'---------

```
,---PARTITIONS-----------------------------------------------------.
|  Q                                                              | <-[ `rA Q` is the same as just `r
|::::::::::::::::::::::::::::::  D                                |
|  Z            ::::::::::::::::::::::::::::::::::::::::::::::::::|
|:::::::::::::::  Y            ::::::::::::::::::::::::::::::::::|
|:::::::::::::::::::::::::::::  X             :::::::::::::::::::|
|:::::::::::::::::::::::::::::::::::::::::::  W                   |
|:::::::::::::::::::::::::::::::::::::::::::  HI    :::::::::::|
|:::::::::::::::::::::::::::::::::::::::::::::::::  LO          |
'-----------------------------------------------------------------'
```

opcode	arguments			example
byte	byte	byte	byte	
NOP	0	0	0	NOP
LDS	dest	short literal		LDS rA 0xFACE
LDA	dest	addr	off	LDA rA rB rC
LDI	dest	addr	lit	LDI rA rB 0x8
STS	src	short literal		STS rB 0xCAB
STA	src	addr	off	STA rB rC rD
STB	src	addr	lit	STB rB rC 0x9
MOV	dst	src	0	MOV rC rD
SWP	dst	src	0	SWP rC rD
ADS	dest	short literal		ADS rD 0xCAFE
SUS	dest	short literal		
MUS	dest	short literal		
DIS	dest	short literal		
MOS	dest	short literal		
ANS	dest	short literal		
ORS	dest	short literal		
SLS	dest	short literal		
SRS	dest	short literal		
XRS	dest	short literal		
ADI	dst	right	0	ADI rD rE
SUI	dst	right	0	
MUI	dst	right	0	
DII	dst	right	0	
MOI	dst	right	0	

```
|  ANI     |  dst   |  right |  0     |                  |
|  ORI     |  dst   |  right |  0     |                  |
|  SLI     |  dst   |  right |  0     |                  |
|  SRI     |  dst   |  right |  0     |                  |
|  XRI     |  dst   |  right |  0     |                  |
|  ADD     |  dst   |  left  |  right |   ADD rD rE rF   |
|  SUB     |  dst   |  left  |  right |                  |
|  MUL     |  dst   |  left  |  right |                  |
|  DIV     |  dst   |  left  |  right |                  |
|  MOD     |  dst   |  left  |  right |                  |
|  AND     |  dst   |  left  |  right |                  |
|  OR      |  dst   |  left  |  right |                  |
|  SHL     |  dst   |  left  |  right |                  |
|  SHR     |  dst   |  left  |  right |                  |
|  XOR     |  dst   |  left  |  right |                  |
|  INV     |  dst   |  src   |  0     |   INV rE rF      |
|  COM     |  dst   |  src   |  0     |   COM rE rF      |
|  INI     |  dst   |  0     |  0     |   INI rF         |
|  COI     |  dst   |  0     |  0     |   COI rF         |
|  CMP     |  left  |  right |  0     |   CMP rA rB      |
|  CMS     |  left  |  short literal  |   CMS rA 0xBEEF  |
|  RET     |  tar   |  0     |  0     |   RET rB         |
|  REI     |  0     |  0     |  0     |   REI            |
|  RES     |  short literal  |  0     |   RES 0xDEAD     |
|  CAL     |  0     |  0     |  0     |   CAL            |
|  PSH     |  tar   |  0     |  0     |   PSH rC         |
|  PSS     |  short literal  |  0     |   PSS 0xAFFE     |
|  POP     |  dst   |  0     |  0     |   POP rD         |
|  BNC     |  mode  |  short literal  |   B**            | --.
|  BNE     |  mode  |  short literal  |   B** label      |   |
|  BEQ     |  mode  |  short literal  |   B** .sub       |   |      ,--MODES--
|  BLT     |  mode  |  short literal  |   B** 0xB        |   |      |
|  BGT     |  mode  |  short literal  |   B** -0xC       |   |      | 0: jump by relative
|  BLE     |  mode  |  short literal  |   B** LR         |   |      |    register stored
|  BGE     |  mode  |  short literal  |                  |   :------:    offset
|  JMP     |  mode  |  short literal  |   J**            |   |      | 1: jump by relative
|  JNE     |  mode  |  short literal  |   J** label      |   |      |    short literal
|  JEQ     |  mode  |  short literal  |   J** .sub       |   |      |    offset
|  JLT     |  mode  |  short literal  |   J** 0xD        |   |      | 2: jump to register
|  JGT     |  mode  |  short literal  |   J** -0xF       |   |      |    stored addtess
|  JLE     |  mode  |  short literal  |   J** LR         |   |      |
|  JGE     |  mode  |  short literal  |                  | --'      '---
|  EXT     |  call  |  0     |  0     |   EXT OUT        |
|  EXR     |  reg   |  0     |  0     |   EXR rE         |
'----------'--------'--------'--------'------------------'
```

A simple program in this language looks about how you would expect.

```
main:
    "hello world\n"
```

```
    MOV rA SP
    ADS rA 0x1
    LDS rB 0xC
    EXT OUT
    REI
```

An abstraction exists to call procedures and an abstraction exists to write raw bytes to the progrma file itself, denoted by wrapped tokens in parenthesis and curly braces respectively.

```
(procedure 0xCAFE rA {0xDEAFBEEFFACECAFE})

;evaluates to

CAL
PSS 0xCAFE
PSH rA
{0xDEADBEEFFACECAFE}
BNC procedure
```

The macro system was written in about an hour as an afterthought, but it was the first attempt at a direction for my language projects to be truly self programmable, I had the idea that it would be really convenient if I had the ability to write high level programs on the level of haskell, while also being able to fine tune performance on the level of assembly. I envisioned this language being bare bones with the ability to build any langauge feature from the building blocks you are provided in the language yourself. Thinking not too much of it at the time, I output the following scheme.

```
; definition

if condition consequent = [
    (condition)
    POP rA
    CMS rA 0
    JEQ 0x3
    CAL
    BNE consequent
]

; use


[if (less rA rB) exit]
```

This continued to spark thought in my mind about what a true metacomputive environment would look like.

4 Prototext

Forthlike languages are so simple to make that this language, written again as a simple exercise and exploration into forthlikes, lands at a cute 1300 lines of C. Its spec is very simple.

```
,----Stack Management.---Label Jumps-------------,
| dup  | a b c c     | jmp  | jeq  | jne          |
| pop  | a b         | jtr  | jfa  | jlt          |
| ovr  | a b c b     | jgt  | jge  | jle          |
| swp  | a c b       :---Word Management----------:
| nip  | a c         | ret  | return from proc    |
| rot  | b c a       | :    | last token is word  |
| cut  | b c         | ,    | last token is label |
:----Change Mode-----: ;    | next token is label |
| u8   | i8          |      | reference           |
| u16  | i16         | .    | exec last token     |
| u32  | i32         :---Example-----------------:
| u64  | i64         | ( these are comments )     |
:----ALU------------| start. (jump to start)     |
| and  | add  | shr  |                            |
| or   | sub  | shl  | start: "hello world        |
| xor  | mul  |      | " loop, u64 0xb u8 1 write |
| not  | div  |      | ;loop jmp ret.             |
| com  | mod  |      |                            |
:----stdout----------:---------------------------:
| write | expects [length fileno] <- top of stack |
`-------------------'---------------------------'
```

Every language I write comes from a desire for my languages to be performant. As an experiment I had this language compile to C rather than be interpreted, and the specification reflects C in its type system, and in that there are no word defintions, rather there are jumpable labels for control flow. I wanted it to be as close to assembly as possible while still being forthlike, forsaking the purity of the classic word defintion.

```
start.
start: "hello world
" u64 12 u8 1 write ret
```

I had a thought to return to forth for its purity, while I dont believe pure systems are the future, I do appreciate systems which achieve a lot with very little.

5 Ink

Ink is the most serious attempt at a vialbe language project. It was abandoned when I decided that I didnt want a static language as my computation sandbox, it was too restricting. The idea was to have a computation medium that reflected how I think about programs, and what I thought a perfect language could look like. I now know that I missed the mark, partiularly in the rigor of the type system, its more

rigorous than C, but not nearly constrictive enough for the modern age. Its type inference is actually quite sophisticated. Other than the few compiler bugs that do exist as it is an abandoned project, its an entirely usable compiler and language. This as also the first project I did error reporting in, and its lackluster. I learned a lot from this project and could likely do a much better job today, but I dont think I will return to static compiler programs as I have decided to focus on metacompilation instead, where compilation is the medium of computation.

It is worth noting that Ink compiles to C.

An Ink program consists of a list of term, constant, type, and type alias definitions. The order of these definitions is arbitrary, as Ink will resolve type dependencies.

A term requires a type, a name, and a value.

```
u64 x = 5;
```

A type or alias require a name and a type.

```
type byte = i8;

alias cstr = i8^;
```

Constants require a name and a value.

```
constant PI = 3.14;
```

All ink programs require a main term of type u64.

```
import "std/io.ink"

u64 main = print "Hello World!\n";
```

Global terms are computed at reference, local terms or closures are computed at definition.

```
// gives you a single line comment

/*
    gives you a multi line comment
*/
```

Types may be one of a limited set of primitives.

Unsigned integers: u64, u32, u16, u8

Signed integers: i64, i32, i16, i8 [] Single and double precision floating point numbers: f32, f64

Types may be pointers to other types, a pointer to a type A looks like A^. Ink supports native fat pointers, containing a pointer and a length, a fat pointer to A looks like [A]. Types may also represent functions, a function taking an argument A and returning a type B looks like A $->$ B. A function to a pointer type like A $->$ B^may be written with the shorthand A $>$ B. All function types are left associative. The last three compound types are structure types, and accord precisely to their C language equivalents. struct is a product type.

```
type pair = struct {
    u64 a;
    u64 b;
};
```

you can also make packed structures with pack.

```
type pair = pack{
    u64 a;
    u64 b;
};
```

Instances of structs can be created and accessed like

```
u64 main = {
    pair x = {3, 4};
    pair y = {b=3, a=4};
    pair z = {a=3, 4};
    u64 position_x = x.a;
    u64 position_y = y.b;
    return position_x;
};
```

Fat pointers are structures internally, and their members may be accessed as ptr and len. The length is stored as a u64. union is the data component of a sum type.

```
type entity = union {
    struct {
        f32 x;
        f32 y;
        u8 health;
    } player;
    struct
        f32 x;
        f32 y;
        entity^ target;
    } enemy;
};
```

Unions may be accessed just like structures. Unions are not tagged by default, but may be informally tagged with enumerators. enum is a primitive enumerator. Its enumerated values are treated as literals, and its internal values begin at 0, and increment linearly unless explicitly defined.

```
type entity_type = enum {
    PLAYER_ENTITY = 1,
    ENEMY_ENTITY  // continues incrementing, its value is 2
};

alias Bool = enum {
    false, // 0
    true   // 1
};

u64 main = {
    Bool happy = true;
    Bool tired = false;
    return 0;
};
```

All types are infinitely composable. Keep in mind that alias definitions cannot be recursive at all (even pointer recursive), and structure/union types are not allowed to be structure or union recursive. You can cast between really any two types with the as keyword.

```
u64 x = 0;
u8 y = x as u64;
```

You can get the size of any type interface with the sizeof keyword.

```
u64 x = sizeof struct {u64 a; u64 b;};
```

All types are immutable by default, but can be marked mutable with the var keyword. Mutation may only happen explicitly, no implicit mutation function exist such as += -= ++ −. Mutation can of course only be done on terms marked with var . Global terms are computed on reference and can not be mutated, local terms are computed at declaration and can be mutated. Mutation is performed with the = special form.

```
u64 main = {
    f32 var x = 0;
    x = 1;
    return x;
};
```

Primitives

```
u8 a = 0;
i8 b = -1;
f32 c = 9;
f32 d = 3.14;
f32 e = -3e10;
f64 f = 3.8e-9;
```

More complicated expressions exist for each of the compound types. We have already seen structure constructors, but there is special syntax for unions

```
type entity = struct {
    enum {PLAYER, ENEMY} tag;
    union {
        struct {
            f32 x;
            f32 y;
            u8 hp;
        } player;
        struct {
            f32 x;
            f32 y;
            entity^ target;
        } enemy;
    };
};

u64 main = {
    struct {u64 a; u64 b;} pair = {a=0, b=1};
    struct {u64 a; u64 b;} pair2 = {0, 1};
    struct {u64 a; u64 b;} pair3 = {a=0, 1};
    struct {u64 a; u64 b;} pair4 = {5}; // sets a
    struct {u64 a; u64 b;} pair5 = {b=0, a=1};

    entity bob    = {PLAYER, player = {0, 0, 10}   };
```

```
    entity zombie = {ENEMY,  enemy  = {0, 0, &bob} };

    return 0;
};
```

Pointers and fat pointers may be assigned to list or string literals.

```
u64 main = {
    i8^ cstring = "Hello world\n";
    [i8] fat_cstring = "Hello world\n";

    u8^ x = [1, 2, 3];

    [u8] y = [ // fat pointer to a buffer of length 3
        [0], 3
    ];

    return 0;
};
```

Pointers may also be set to the address of another term.

```
u64 main = {
    u32 x = 0;
    u32^ ptr = &x;
    [u32] = [&x, 1];
    return ^ptr;
};
```

Blocks are multi-line expressions. Since all expressions in Ink are ; terminated, block expressions simply require sequential expressions. The final expression in a block must be a return expression, which designates the value of the block expression if it were to be computed. All expressions in a block may use all previous term definitions in the block to create compound expressions.

```
u64 main = {
    u64 x = 0;
    u64 y = {
        u64 f = x + 6;
        return f
    };
    return y;
};
```

If/else can be used as both a statement and an expression, if used as an expression it must have an else and the resulting types between the two branches must match, otherwise they can have any type and dont have to return. If used as a statement, returning will return from the outer block.

```
u64 main = {
    u64 x = if 1 { return 8; } else { return 9; };
    if x {
        return 1;
    };
    return 0;
};
```

While is just repeated if, you cannot use it as an expression and it doesn't get an else.

```
import "std/io.ink"
import "std/ffi.ink"

u64 main = {
    u64 var i = 0;
    while i < 10 {
        print "Hello\n";
    };
    return i;
};
```

For is a structured while loop, it takes an initial clause, a condition to check, and a mutation clause that runs at the end of each iteration.

```
import "std/io.ink"
import "std/ffi.ink"

u64 main = {
    for i32 var i = 0; i<10 ; i=i+1 {
        print "hi\n";
    };
    return 0;
};
```

This is a pattern matching statement.

```
type Maybe = struct {
    enum {Just, Nothing} tag;
    i32 val;
};

Maybe -> u8 f = \m:
    match m {
        (Just 1): 0;
        (Just x): x as u8;
        result@(Nothing): result.val;
        _: 1;
    };
```

Patterns follow the following composable rules:

binding@inner_pattern names a pattern and continues to match against it

[fat_ptr_ptr fat_ptr_len]

_ matches anything

(x 6 5) matches a structure with the first member bound to x for the resulting expression, and the next two members checked against 6 and 5

All functions in Ink are values, the canonical value of a term with a function type, is a lambda expression. Lambdas have arguments and a resulting expression.

```
u64 -> u64 incrememnt = \x: x + 1;

u64 -> u64 -> u64 sum = \a b: a + b;
```

```
u64 -> u64 -> u64
greater = \left right: {
    u64 x = left > right;
    return left > right;
};
```

Lambda args are patterns, and will be matched before the function starts. You may consequently have alternate matching cases for lambda args:

```
Maybe -> u8 f = \(Just x): x as u8;
              | \(Nothing): 0;
              | \_: 0;
```

To apply a function, we simple place its argument(s) after its name. Symbol named functions are allowed to be infix but may also be prefixed.

```
u64 -> u64 -> u64 sum = \a b: a + b;

u64 main = {
    u64 x = sum 4 5;
    u64 y = (- 10 x);
    return x + y;
};
```

You do not need to apply function fully, they may instead be partially applied.

```
u64 -> u64 -> u64
sum = \a b: a + b;

u64 main = {
    u64 -> u64 add_four = sum 4;
    u64 y = add_four 5;
    return y;
};
```

Just like in Haskell, you can use $ to turn the default left associative function application into a right associative one.

```
// these two are equivalent
f $ x y z;
(f ((x y) z));
```

Composition has a syntactic shorthand as well.

```
// these two are equivalent
f . g . h a;
(f (g (h a)));
```

The body of a term will not be evaluated by an application to a reference to that term until that term's type is reduced to a non function. It should be mentioned that lambdas, structure expressions, blocks, and any other type of expression, can be used entirely anonymously as long as the type is inferrable from its context.

Memory stored in a memory space other than the managed stack region, must be managed explicitly. This follows the semantics of languages like C, mostly this means you just shouldn't dangle stack pointers

to stack allocated terms, but the way closures are represented in Ink requires special attention to the memory of the program. Closures in Ink are designed to ensure the programmer still has control over the way their memory is being allocated, and the way its being accessed. You can safely use closures knowing predictably how their metadata will be accessed and stored. This means you can reason about functional programs using closures without having to rely on the slowdowns of default heap allocation, odd memory access patterns causing cache misses, or garbage collection overhead. All you need to know is that closures are secretly fat pointers to stack allocated packed structures. Since they are stack allocated, this means you can pass closures as arguments to terms without issue.

```
u64 -> u64 -> u64 sum = \x y : {
    return x + y;
};

(u64 -> u64) -> u64 -> u64
mutator = \mutation value:{
    return mutation value;
};

u64 main = {
    u64 -> u64 inc = sum 1;
    u64 seven = mutator inc 6;
    return 0;
};
```

Returning them however, requires you move their memory from the stack to an alternative memory region with a custom allocator. The tiny standard library which does exist for Ink includes an arena allocator, which implements the Allocator typeclass.

```
import "std/io.ink"

arena^ -> u64 -> u64 -> u64 incrementor = \a x: {
    u64 -> u64 f = (\y: x + y);
    print "side effect\n";
    return f =:>> a; // moves f into a
};

u64 main = {
    arena pool = arena_init 100;
    u64 -> u64 inc = incrementor (&pool) 1;
    print "After first application\n";
    u64 three = inc 2;
    return 0;
};
```

Note here that the output order is

```
After first application
side effect
```

This is because the applied reference to incrementor has not resolved to a non function type, so the procedure does not evaluate until that type is reduced at the operation.

```
u64 three = inc 2;
```

If you want explicit currying, that is, returning a closure while running the procedure it came from, you can do with by returning a pointer to that location.

```
import "std/io.ink"

arena^ -> u64 ~> u64 -> u64 effectful = \a x: {
    u64 -> u64 f = \y: x + y;
    print "side effect\n";
    return f =:>> a =:> a; // moves f to a, and creates a pointer to that region in a
};

u64 main = {
    arena pool = arena_init 100;
    u64 -> u64 f = ^(effectful (&pool) 1);
    u64 three = f 2;
    print "After effect\n";
    return 0;
};
```

This gives Ink very explicit effect semantics, allowing an information pointer based notation for when partial effects happen, understanding that otherwise, side effectful computation can only occur at full application to a term.

Types may be parametric, and this can be notated in type definitions. This may also be a good time to mention that you can define functions with symbol names, these can be infix or prefix in use.

```
//a generic pipe procedude
A -> (A -> B) -> B
|> = \val fun: fun val;

type Either L R = struct {
    enum {Left, Right} tag;
    union {
        L left;
        R right;
    } data;
};

u64 main = {
    Either u8 i8 applied_generic = {Left, left = 0};
    return applied_generic.data.left as u64;
};
```

An emergent behavior of the Ink typechecker has lead to an implicit let type inference feature, where because Ink will try to evaluate the type of a generic non function type and replace the definition with the evaluated type, you are allowed to use any arbitrary generic term not otherwise used in the scope to do implicit type inference. This can only work if the type is actually deducible with all the other type information present.

```
u64 main = {
    u64 x = 0;
    let y = x + 6; // Ink knows y is u64
```

```
    return y;
};
```

Typeclasses are sometimes called interfaces, or protocols, or traits, but I was first introduced to them while programming Haskell, so to me they are typeclasses. I don't know what the original literature for this feature has to say about its naming. Typeclasses need a name and a member parameter which represents a generic instance of that typeclass in subsequent member term definitions.

```
typeclass Functor F {
    (A -> B) -> F A -> F B map;
}
```

Implementations then replace this generic term with the type implementing the typeclass. It should be noted that only named types can have typeclass instantiations.

```
type buffer T = [T];

byte_list implements Functor {
    (A -> B) -> buffer A -> buffer B
    map = \f data: {
        u64 var i = 0;
        buffer (B var) result = [ [0], data.len ];
        while i < data.len {
            result.ptr[i] = f (data.ptr[i]);
            i = i + 1;
        };
        return result;
    };
}
```

Generic types in any term may have a typeclass dependency.

```
(Allocator A) => A -> T -> T^ mk_ptr = \alloc data: ...;
```

Typeclass members have the typeclass parameter distributed over them as a dependency.

```
typeclass Orderable O {
    O -> O -> i8 compare;
}

//is converted internally to

typeclass Orderable O {
    (Orderable O) => O -> O -> i8 compare;
}
```

Ink has the ability to completely interface with C, there may be friction in some places, but you can run arbitrary C from Ink and arbitrary Ink from C. To define external terms, aliases, or types, declare them in an external block.

```
external {
    u8^ -> u8^ -> u64 -> u8 memcpy;
    u8^ -> u8 -> u64 -> u8 memset;
    u64 -> u8^ -> u64 -> u64 write;
    u64 -> u8^ malloc;
}
```

Note that typedef struct name {} name in C translates approximately to type name = struct {} in Ink, but any other non structure typedef is equivalent to an alias in Ink, not a type definition. To import local C files, you may import normally.

```
external {
    import "emscripten_wrapper.h"
}
```

To import C standard library files, you may import globally.

```
external {
    import global "SDL2/SDL.h"
}
```

Some FFI interfaces have already been generated for SDL2, netinet/in, and emscripten. Enumerators and constants are not handled well by the FFI generator, so some of them have been defined manually as constants in Ink.

There are many examples of full ink programs in the git repository, as well as the beginning of a standard library, and some simplified examples at the end of the README. When writing this document I was reminded of the level of sophistication I reached during this project that I had all but forgotten about. The views which created ink are no longer explicitly the views I hold about programming, I've learned a lot and grown out of those shoes. You can still see artifacts of my beliefs in ink, such as where I try to unify systems, and try to find patterns which generalize.

6 Src

Src (source) went through a bunch of iteration passes which have been lost to commit history. It started as a more serious attempt at a programmable abstraction programming language where you build up every abstraction and have complete control over every level, essentially writing a compiler as you write your program. Initially there was a very specific metalanguage layered ontop of it which went through a few iteration fo its own before dissapearing entirely and being replaced with a pure bytecode metacomputation pass system which takes in the entire program as an input and performs symbolic mutation to produce a subsequent program.

While exactly half the code size of Ink, it's less bug prone and equally as sophisticated, featuring a high performance multicore virtual machine which you can layer and perform arbitrary compile time execution with at any time.

I used this staged environment based persistent compile time layering system in the systems direct successor, un. I learned from this project that the simpler a language the more malleable and manageable it is. This precludes any software I suppose. The trick with writing metacompileres is that you are really only writing half a language, the other half being any given program you write. This makes it easier to implement full languages using the metacompiler as a backend.

Source is a metacomputation tool. It can serve as a standalone virtual computer, but can be used as a plugin manager for basically any base language. You define syntactic mutations to your file in a well defined language close to how real computation happens on real machines, but a little more abstract.

You can then define semantic state for those syntactic constructs in the same language. A runtime environment exists alongside the compile time environment, of course also in the same language, which can be used to write any real software you want, but this tool is mainly meant to build headless extensions that can be tacked onto base languages. True unbounded freedom, allowing absolute precision. Ever want assembly with a proof engine tacked on? Now you can make it. You can run multiple instances of source computers in the same directory and have them communicate over an emulated network via file io.

Passes handle the syntactic side of metaprogramming. They operate on the program file memory itself inside a vm. r0 is given the address of the source program, r1 is the address of the start of the output program, which replaces your program when the pass runs. The following defines double slash comments.

```
pass
mov 0 r1
bind continue ip
    mov r2 r0
    movw r3 !/
    cmp [r0] r3
    jne append
    add r0 r0 !8
    cmp [r0] r3
    jne append
    bind inner ip
        add r0 r0 !8
        movw r3 !\
        cmp [r0] r3
        jne inner
        add r0 r0 !8
        jmp continue
    bind append ip
    mov r0 r2
    mov [r1] [r0]
    add r1 r1 !8
    add r0 r0 !8
    cmp r0 0
    jlt continue
bind break ip
mov r0 !1
int
end
// then you can use the abstraction like this!
```

Comptime handles the semantic side of metaprogramming. It gives you a compile time vm to store persistent data accross comptime blocks.

```
comp
    anything here runs at comptile time
    mov r0 !1
    int
run
```

Binds allow you to move values from compile time to runtime.

```
comp
    mov r1 !FF
```

```
    mov r0 !1
    int
run

bind  hi_byte = r1
```

binds hi_byte to the constant stored in r1 on the comptime vm: FF.

```
    unbind hi_byte
```

This releases the binding. The following is a grammar for the interpreted instruction set.

```
Builtin Regsiters = r0 | r1 | r2 | r3 | ip
Opcodes = mov loc loc
        | movw loc locw
        | movh loc loc
        | movl loc loc
        | add loc loc loc
        | sub loc loc loc
        | mul loc loc loc
        | div loc loc loc
        | mod loc loc loc
        | and loc loc loc
        | xor loc loc loc
        | or loc loc loc
        | shl loc loc loc
        | shr loc loc loc
        | not loc loc
        | com loc loc
        | cmp loc loc
        | jmp loc
        | jlt loc
        | jgt loc
        | jeq loc
        | jne loc
        | jle loc
        | jge loc
        | int

loc = [deref] | literal32 | address
locw = literal64
deref = literal32 | address
address = (hex integer)
literal = !(hex integer)
```

The following is a list of interrupts and their register expectations.

```
r0: 0
blit frame buffer to screen

r0: 1
end program execution

r0: 2
```

```
r1: value
print value to external stderr

r0: 3
r1: key
r2 <- is key down

r0: 4
r1: key
r2 <- is key pressed

r0: 5
r1 <- mouse x
r2 <- mouse y

r0: 6
r1: mouse button
r2 <- is button down

r0: 7
r1: mouse button
r2 <- is button pressed

r0: 8
r1: in address of program
r2: in length of program
r3 <- address to write compiled program

r0: 9
r1: instruction pointer for new core to start at
r0 <- core running, 0 for no core available

r0: a
r1: file network address word
r2: packet start address
r3: packet length (bytes)
send message to another src computer

r0: b
r1: address to dump network addresses
r2 <- number of addresses found

r0: c
r1: address to read message into
r2 <- length of message
```

There are also a number of builtin symbols for helping with some of the interrupts.

```
mtp: memory top
mbm: memory bottom (frame buffer_size)
fbw: frame buffer width
fbh: frame buffer height
SRC_MOUST_LEFT
```

```
SRC_MOUSE_RIGHT
SRC_MOUSE_MIDDLE
SRC_Q - M
SRC_LEFT RIGHT UP DOWN
SRC_SPACE
```

There are a number of examples of src in the git repository. An unfortunate design decision made to try to unify the memory model as far as possible lead to registers being normal memory locations, so that you could easily define your own registers. The stack for instance is not baked into src, instead allowing the user to define if they do or do not want to use stack based computation, or any other paradigm (Forthlikes are remarkably simple to do, and can be accomplished in couple hundred lines of src, complete with word definitions). This however lead to multicore capabilities experiencing friction where you need to define a register set for every core, and make sure that each subprocess switches to its correct register set before execution. This makes generalized core agnostic code possible, but a bit of a headache to setup and manage. This mistake was not repeated in the virtual machine for un.

7 un

Un is, at the time of drafting this document, the present iteration of my metacomputation tooling. It takes a more serious look at the offhanded thought I had at the beginning of the year, mentioned at the beginning of this document: a lisp which manages and outputs assembly. In reality its a little more complicated than that, and needs to be in order to be a tool for the real world.

7.1 A Note on Lisps

Lisp is not a paradigm, you coudl argue that only a language with Lisp style macros is a true Lisp dialect, but Lisp otherwise just refers to a language which uses S Expressions as a syntactic form. Otherwise, un is not a Lisp. It does not have Lisp style macros, although it achieves the same result through a combination of atomic abstractions which deliniate naming, and computation separately. I used S expressions, but I do not call this a Lisp.

7.2 Pieces of a Complete Metacomputation Environment

Un differs in architecture from src in a few key ways. First, its virtual machine instruction set is more like what I real computer uses. The interpreted nature of these systems I need to note, is temporary and is a consequence of iteration. Ultimately the bytecode is simple enough that it can be trivially translated to any popular real bare metal ISA.

This means however that un gets core agnostic bytecode in its interpreted mode, and gets to remain multicore. The comptime staging is now more explicit, allowing you to specify which virtual machine to perform computation in at compile time, all distinct virtual machines have persistent memory segments, except for an ephemeral static section before the program section used for symbolic reification at compile time.

Un also, notably, uses a structured metacomputation substrate, S expressions. I noticed while writing src programs that it was truly a chore to get anything off the ground, it was there that I realized there was a cost to full unbridled power. I also noted that I had a system which could constrain itself and output errors on user defined compilation stops. However I wanted to explore this more in depth. For un, the structured metacompuation substrate allows you to define custom constraint systems and import them like libraries. This is in the second portion of language writing, where metacompilers give

you the first half, the second half is responsible for giving these constraint systems, which are typically the interesting part of compiler research. With un, I no longer have to write entire systems to test out ideas or try to break invariant systems.

One pain point still is that S expressions are also a chore to write, especially S expressions describing bytecode, which is a lot of the initial setup to any un project. S expressions as an abstraction layer certainly offset the arduous task of writing raw bytecode all the time, but it's still not ideal. As someone who programs a lot, I know how writing in different languages feels, and I know its important. I dont think syntax matters for the sake of a metacomputation tool, but I do think it matters in languages facing the real world. The idea is to research languages using un as a sandbox, and when good designs emerge as a result of the research, I can write another simple parser for a more user friendly syntax, making un like a language writing factory.

As un is still in active development, documentation for it will be in a different note entry. Currently register coloring is broken, but the present architecture outputs an intermediate representation which is then normalized and flattened into static bytecode. Un is looking like it will be an even smaller implementation effort than src was, while being tantamount in sophistication.

8 Direction

From this first year of preliminary prototyping and research a clear direction has emerged. Lisp meta compilers like Meta II failed because of a lack of real compiler like support, including systems of constraint and error. These systems are supportable from within the un language itself as a protocol which sublangauges can support. Since these constraint systems will be expressed programmatically they can be generated dynamically in order to mix invariant sets and find interesting constraint systems. Through guided program synthesis we can generate effectful programs which mimic the patterns and actions of real world internet facing software while pertaining to the rules of a given constraint system and then adversarially engage these programs to find cases where systems break under their own invariant sets.