C++ is a Metacompiler

Daniel Nikpayuk

2024

Outline

Outline

1. Introduction

Outline

1. Introduction
2. Demonstration

Outline

Outline

1. Introduction
2. Demonstration
3. Philosophy
4. Methodology

Outline

Outline

Outline

Outline

Introduction

Who am I?

- I'm a self-taught coder.
- I've been programming in C++ since 2005.
- I don't currently work in the tech industry.
- I have a Bachelor of Arts majoring in mathematics, minoring in economics.
- I am an Inuit person (specifically Inuvialuit) from Canada's western Arctic.
- I am devoted to the continued renewal of my people's language and culture.

Why C++?

- It is a life goal of mine to build a programming language for multimedia production.
- I hope to offer said language as an option for telling and retelling my people's stories, traditional and new.

Figure: inuksuk

- Such a language will generally require systems level performance, and so C++ is a good fit for writing its first compiler.

Demonstration

- What is a metacompiler?

- What is a metacompiler?
- The Philosophy section discusses this design.

- What is a metacompiler?
- The Philosophy section discusses this design.
- Here we show a few code examples:

Example 1: Factorial

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "  test equal n 0      ;"
                "  branch done         ;"
                "  . = subtract n 1    ;"
                "  . = factorial _     ;"
                "  . = multiply n _    ;"
                "  return _            ;"

                "done:                 ;"
                "  return 1:T          ;"
        );
}
```

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "  test equal n 0      ;"
                "  branch done         ;"
                "  . = subtract n 1    ;"
                "  . = factorial _     ;"
                "  . = multiply n _    ;"
                "  return _            ;"

                "done:                 ;"
                "  return 1:T          ;"
        );
}
```

chord language, factorial:

Example 1:

- A C++17 constexpr function.

chord language, factorial:

Example 1:

- A C++17 constexpr function.
- This function takes no input.

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "   test equal n 0     ;"
                "   branch done        ;"
                "   . = subtract n 1   ;"
                "   . = factorial _    ;"
                "   . = multiply n _   ;"
                "   return _           ;"

                "done:                 ;"
                "   return 1:T         ;"
        );
}
```

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                 ;"
                "factorial n -> T       ;"

                "body:                  ;"
                "   test equal n 0      ;"
                "   branch done         ;"
                "   . = subtract n 1    ;"
                "   . = factorial _     ;"
                "   . = multiply n _    ;"
                "   return _            ;"

                "done:                  ;"
                "   return 1:T          ;"
        );
}
```

chord language, factorial:

Example 1:

- This function returns “source” data.

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "   test equal n 0     ;"
                "   branch done        ;"
                "   . = subtract n 1   ;"
                "   . = factorial _    ;"
                "   . = multiply n _   ;"
                "   return _           ;"

                "done:                 ;"
                "   return 1:T         ;"
        );
}
```

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "  test equal n 0      ;"
                "  branch done         ;"
                "  . = subtract n 1    ;"
                "  . = factorial _     ;"
                "  . = multiply n _    ;"
                "  return _            ;"

                "done:                 ;"
                "  return 1:T          ;"
        );
}
```

chord language, factorial:

Example 1:

- The source data is a string literal.

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                 ;"
                "factorial n -> T       ;"

                "body:                  ;"
                "   test equal n 0      ;"
                "   branch done         ;"
                "   . = subtract n 1    ;"
                "   . = factorial _     ;"
                "   . = multiply n _    ;"
                "   return _            ;"

                "done:                  ;"
                "   return 1:T          ;"
        );
}
```

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "   test equal n 0     ;"
                "   branch done        ;"
                "   . = subtract n 1   ;"
                "   . = factorial _    ;"
                "   . = multiply n _   ;"
                "   return _           ;"

                "done:                 ;"
                "   return 1:T         ;"
        );
}
```

chord language, factorial:

Example 1:

- The string literal is source from another language.

chord language, factorial:

Example 1:

- The string literal is source from another language.
- This other source is a domain specific language (DSL).

chord language, factorial:

Example 1:

- The string literal is source from another language.
- This other source is a domain specific language (DSL).
- This DSL is an assembly inspired language called chord.

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "   test equal n 0     ;"
                "   branch done        ;"
                "   . = subtract n 1   ;"
                "   . = factorial _    ;"
                "   . = multiply n _   ;"
                "   return _           ;"

                "done:                 ;"
                "   return 1:T         ;"
        );
}
```

chord language, factorial:

Example 1:

- Relevance?

chord language, factorial:

```
using factorial_v0 = chord::metacompile
<
        _chord_factorial_v0,
        null_env, unsigned long
>;

static_assert(factorial_v0::result(9) == 362880);
```

chord language, factorial:

```
using factorial_v0 = chord::metacompile
<
        _chord_factorial_v0 ,
        null_env , unsigned long
>;

static_assert(factorial_v0::result(9) == 362880);
```

chord language, factorial:

In effect:

- We metacompile our DSL source code.

chord language, factorial:

In effect:

- We metacompile our DSL source code.
- We pass it as a template parameter.

chord language, factorial:

In effect:

- We metacompile our DSL source code.
- We pass it as a template parameter.
- It returns as a constexpr function.

chord language, factorial:

```
using factorial_v0 = chord::metacompile
<
        _chord_factorial_v0,
        null_env, unsigned long
>;

static_assert(factorial_v0::result(9) == 362880);
```

chord language, factorial:

```
using factorial_v0 = chord::metacompile
<
        _chord_factorial_v0,
        null_env, unsigned long
>;


static_assert(factorial_v0::result(9) == 362880);
```

chord language, factorial:

- Our metacompiled function can be used either at compile time or run time.

For those who want a bit more detail. . .

For those who want a bit more detail. . .

A metacompiler turns this:

For those who want a bit more detail...

A metacompiler turns this:

```
constexpr auto _chord_factorial_v0()
{
	return source
	(
		"type T                ;"
		"factorial n -> T      ;"

		"body:                 ;"
		"   test equal n 0     ;"
		"   branch done        ;"
		"   . = subtract n 1   ;"
		"   . = factorial _    ;"
		"   . = multiply n _   ;"
		"   return _           ;"

		"done:                 ;"
		"   return 1:T         ;"
	);
}
```

into this:

into this:

```
constexpr size_type value[][8] =
{
    { AN::id        , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::hash      , AT::port      ,  5,  0,  0,  0,  0,  1 },
    { AN::pad       , AT::select    ,  0,  1,  0,  0,  0,  1 },
    { AN::pad       , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::go_to     , AT::id        , 50,  0,  0,  0,  0,  1 },
    { AN::id        , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::eval      , AT::back      ,  7,  0,  0,  0,  0,  4 },
    { AN::id        , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::lookup    , AT::first     ,  0,  7,  0,  0,  0,  1 },
    { AN::halt      , AT::first     ,  0,  0,  0,  0,  0,  1 },
    { AN::eval      , AT::back      , 11,  0,  0,  0,  0,  5 },
    { AN::id        , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::arg       , AT::select    ,  1,  0,  0,  0,  0,  1 },
    { AN::arg       , AT::drop      ,  0,  0,  0,  0,  0,  1 },
    { AN::halt      , AT::first     ,  0,  0,  0,  0,  0,  1 },
    { AN::type      , AT::n_number  ,  0,  0,  0,  0,  0,  1 },
    { AN::literal   , AT::back      ,  0,  0,  0,  0,  0,  1 },
```

which we then pass to this:

which we then pass to this:

```
template<auto... filler>
struct T_assembly<AN::hash, AT::id, filler...>
{
    template<NIK_ASSEMBLY_PARAMS(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
            constexpr auto ni = AD<c>::pos(i);
            constexpr auto nv = U_assembly_compound<c, ni>;

            return NIK_ASSEMBLY_TEMPLATE(c, i)
                    ::NIK_ASSEMBLY_RESULT_2TS(c, i, l, t, r, decltype(nv), Ts...)
                            (nv, vs...);
    }
};
```

which finally turns into a constexpr function.

which finally turns into a constexpr function.

But we're getting ahead of ourselves.

Example 2: Factorial

hustle language, factorial:

```
constexpr auto _hustle_factorial_v0()
{
    return source
    (
        "(type T                              "
        "  (define (factorial n) -> T         "
        "    (if (= n 0)                      "
        "      1:T                            "
        "      (* n (factorial (- n 1)))      "
        "    )                                "
        "  )                                  "
        ")                                    "
    );
}
```

hustle language, factorial:

Example 2:

- We have another C++17 constexpr function.

hustle language, factorial:

Example 2:

- We have another C++17 constexpr function.
- This function also takes no input.

hustle language, factorial:

Example 2:

- We have another C++17 constexpr function.
- This function also takes no input.
- This function again returns “source” data.

hustle language, factorial:

Example 2:

- We have another C++17 constexpr function.
- This function also takes no input.
- This function again returns “source” data.
- The source data embeds another string literal.

hustle language, factorial:

```
constexpr auto _hustle_factorial_v0()
{
    return source
    (
        "(type T                               "
        "  (define (factorial n) -> T          "
        "    (if (= n 0)                       "
        "      1:T                             "
        "      (* n (factorial (- n 1)))       "
        "    )                                 "
        "  )                                   "
        ")                                     "
    );
}
```

hustle language, factorial:

```
constexpr auto _hustle_factorial_v0()
{
    return source
    (
        "(type T                                "
        "  (define (factorial n) -> T           "
        "    (if (= n 0)                        "
        "      1:T                              "
        "      (* n (factorial (- n 1)))        "
        "    )                                  "
        "  )                                    "
        ")                                      "
    );
}
```

hustle language, factorial:

Example 2:

- The string literal encodes source from yet another language.

hustle language, factorial:

Example 2:

- The string literal encodes source from yet another language.
- This source is again a domain specific language (DSL).

hustle language, factorial:

Example 2:

- The string literal encodes source from yet another language.
- This source is again a domain specific language (DSL).
- This DSL is a scheme (lisp) inspired language called hustle.

hustle language, factorial:

```
constexpr auto _hustle_factorial_v0()
{
    return source
    (
        "(type T                          "
        "  (define (factorial n) -> T     "
        "    (if (= n 0)                  "
        "      1:T                        "
        "      (* n (factorial (- n 1)))  "
        "    )                            "
        "  )                              "
        ")                                "
    );
}
```

hustle language, factorial:

Example 2:

- Finally. . .

hustle language, factorial:

```
using factorial_v0 = hustle::metacompile
<
        _hustle_factorial_v0,
        null_env, unsigned long
>;

static_assert(factorial_v0::result(9) == 362880);
```

hustle language, factorial:

Example 2:

- We can metacompile
  and apply this function as well.

Philosophy

- C++ is a metacompiler.

- C++ is a metacompiler.
- We've seen what a metacompiler *does*.

- C++ is a metacompiler.
- We've seen what a metacompiler *does*.
- We now ask what a metacompiler *is*.

Let's take a short tour
of related concepts.

We start by asking:

We start by asking:

- What is a compiler?

We start by asking:

- What is a compiler?
- What is an interpreter?

We start by asking:

- What is a compiler?
- What is an interpreter?
- What is a transpiler?

To keep things simple:

To keep things simple:

- A *compiler* takes source code and translates it into assembly.

To keep things simple:

- A *compiler* takes source code and translates it into assembly.
- An *interpreter* takes source code, translates, then executes it directly.

To keep things simple:

- A *compiler* takes source code and translates it into assembly.
- An *interpreter* takes source code, translates, then executes it directly.
- A *transpiler* takes source code and translates it into the source code of another language.

Do these ideas apply to a metacompiler?

Do these ideas apply to a metacompiler?

- Compiler: Yes, a metacompiler takes source code and translates it into an intermediate assembly.

Do these ideas apply to a metacompiler?

- Compiler: Yes, a metacompiler takes source code and translates it into an intermediate assembly.
- Interpreter: Maybe, a metacompiled function can be executed at compile time.

Do these ideas apply to a metacompiler?

- Compiler: Yes, a metacompiler takes source code and translates it into an intermediate assembly.
- Interpreter: Maybe, a metacompiled function can be executed at compile time.
- Transpiler: Maybe, a metacompiler takes source code and does translate it into C++, but only C++.

So I've called it a metacompiler.

So I've called it a metacompiler.

What makes it "meta?"

So I've called it a metacompiler.

What makes it "meta?"

- The prefix comes from metaprogramming.

So I've called it a metacompiler.

What makes it "meta?"

- The prefix comes from metaprogramming.
- In C++ this means compile time programming.

So I've called it a metacompiler.

What makes it "meta?"

- The prefix comes from metaprogramming.
- In C++ this means compile time programming.
- This includes constexpr time programming,

So I've called it a metacompiler.

What makes it "meta?"

- The prefix comes from metaprogramming.
- In C++ this means compile time programming.
- This includes constexpr time programming, as well as template metaprogramming.

As such, a metacompiler requires
we refine our notion of time.

We ask:

We ask:

- What is a timescape?

We ask:

- What is a timescape?
- What is a timescope?

The short answer:

> When observing the lifespan of a program,
> a timescape allows us to decompose it into timescopes.

As for specific timescopes:

As for specific timescopes:

- Run time is when a program is being executed.

As for specific timescopes:

- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.

As for specific timescopes:

- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.
- Metarun time is when a metaprogram is being executed. . .

As for specific timescopes:

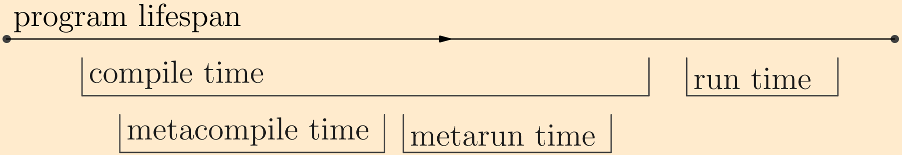
- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.
- Metarun time is when a metaprogram is being executed. . . *within the scope of compile time.*

As for specific timescopes:

- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.
- Metarun time is when a metaprogram is being executed. . . *within the scope of compile time.*
- Metacompile time is when a metaprogram is being translated for execution. . .

As for specific timescopes:

- Run time is when a program is being executed.
- Compile time is when a program is being translated for execution.
- Metarun time is when a metaprogram is being executed. . . *within the scope of compile time.*
- Metacompile time is when a metaprogram is being translated for execution. . . *within the scope of compile time.*


program lifespan
compile time
run time
metacompile time
metarun time

What about constexpr time?

What about constexpr time?

- This is C++ specific.

What about constexpr time?

- This is C++ specific.
- This timescope in effect represents either run time or metarun time.

A metacompiler requires we also consider ideas of self similarity.

Why?

Why?

Because in theory we could metacompile source code from the same language that is otherwise being compiled.

Given this, we ask:

Given this, we ask:

- What is a metacircular evaluator?

Given this, we ask:

- What is a metacircular evaluator?
- What is a self-hosting compiler?

Given this, we ask:

- What is a metacircular evaluator?
- What is a self-hosting compiler?
- What is an abstract machine?

Given this, we ask:

- What is a metacircular evaluator?
- What is a self-hosting compiler?
- What is an abstract machine?
- What is a virtual machine?

A metacircular evaluator:

A metacircular evaluator:

- Starts with interpreted language.

A metacircular evaluator:

- Starts with interpreted language.
- Builds a metacircular library.

A metacircular evaluator:

- Starts with interpreted language.
- Builds a metacircular library.
- Builds a function called an evaluator.

A metacircular evaluator:

- Starts with interpreted language.
- Builds a metacircular library.
- Builds a function called an evaluator.
- This evaluator simulates the language's own interpreter.

A metacircular evaluator:

- Starts with interpreted language.
- Builds a metacircular library.
- Builds a function called an evaluator.
- This evaluator simulates the language's own interpreter.
- This evaluator can execute source code from the same language.

A self-hosting compiler:

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

- An interpreter is allowed to interleave source code translation with execution.

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

- An interpreter is allowed to interleave source code translation with execution.
- A compiler is restricted to modularizing source code translation and execution.

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

- An interpreter is allowed to interleave source code translation with execution.
- A compiler is restricted to modularizing source code translation and execution. It *must* translate first, only then can it execute.

A self-hosting compiler:

- Is the compiled version of a metacircular evaluator.

What's the difference?

- An interpreter is allowed to interleave source code translation with execution.
- A compiler is restricted to modularizing source code translation and execution. It *must* translate first, only then can it execute.
- This creates subtle differences in their respective designs.

An abstract machine:

An abstract machine:

- Is a state machine.

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[\,0 \rightarrow N\,]$:

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[\,0 \to N\,]$:

$$S_{0:\text{begin}} \quad \to \quad S_1 \quad \to \quad \dots \quad \to \quad S_{N:\text{end}}$$

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[\,0 \to N\,]$:

$$S_{0:\text{begin}} \quad \to \quad S_1 \quad \to \quad \dots \quad \to \quad S_{N:\text{end}}$$

- Such states are usually expected to have the same shape.

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[0 \to N]$:

$$S_{0:\text{begin}} \quad \to \quad S_1 \quad \to \quad \dots \quad \to \quad S_{N:\text{end}}$$

- Such states are usually expected to have the same shape. In effect, you can consider them to be a data structure.

An abstract machine:

- Is a state machine.
- It performs computations by passing data through a chain of states $[0 \to N]$:

$$S_{0:\text{begin}} \quad \to \quad S_1 \quad \to \quad \ldots \quad \to \quad S_{N:\text{end}}$$

- Such states are usually expected to have the same shape. In effect, you can consider them to be a data structure.
- These machines are generally given some kind of controller (sometimes source code) to direct their computation.

A virtual machine

A virtual machine (in recent literature):

A virtual machine (in recent literature):

- Is an abstract machine.

A virtual machine (in recent literature):

- Is an abstract machine.
- Has states that represent actual hardware.

A virtual machine (in recent literature):

- Is an abstract machine.
- Has states that represent actual hardware.
- Can be used to simulate hardware on top of actual hardware.

Do these ideas apply to a metacompiler?

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.
- self-hosting compiler: Maybe, in theory we could rebuild C++ itself,

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.
- self-hosting compiler: Maybe, in theory we could rebuild C++ itself, but done at compile time it might not be performant enough to be worth it.

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.
- self-hosting compiler: Maybe, in theory we could rebuild C++ itself, but done at compile time it might not be performant enough to be worth it.
- abstract machine: Yes, such machines underly the implementation design.

Do these ideas apply to a metacompiler?

- metacircular evaluator: Sort of, in theory we can interleave translation and execution to interpret.
- self-hosting compiler: Maybe, in theory we could rebuild C++ itself, but done at compile time it might not be performant enough to be worth it.
- abstract machine: Yes, such machines underly the implementation design.
- virtual machine: Somewhat, in theory optimized state transitions can be designed with hardware in mind.

What then is a metacompiler?

What then is a metacompiler?

- It is a toolchain of related technologies which translate source code into assembly.

What then is a metacompiler?

- It is a toolchain of related technologies which translate source code into assembly.
- In terms of the technologies that make up this chain,

What then is a metacompiler?

- It is a toolchain of related technologies which translate source code into assembly.
- In terms of the technologies that make up this chain, it is the idea of a DSL engine that is most relevant to this talk.

What is a DSL engine?

What is a DSL engine?

It is an abstract machine which translates domain specific languages into assembly.

To finish this section we ask one more question.

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly,

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification,

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification, C++17 and later.

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification, C++17 and later.
- C++17 has an emergence of grammar and rules to support a DSL engine,

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification, C++17 and later.
- C++17 has an emergence of grammar and rules to support a DSL engine, one which is also performant.

To finish this section we ask one more question.

We've discussed the idea of a metacompiler more broadly, so now we specifically ask:

Why is C++ a metacompiler?

- It is a metacompiler because of its specification, C++17 and later.
- C++17 has an emergence of grammar and rules to support a DSL engine, one which is also performant.
- It is independent of vendor implementation.

Methodology

- C++ is a metacompiler.

- C++ is a metacompiler.
- We've seen what a metacompiler *does*.

- C++ is a metacompiler.
- We've seen what a metacompiler *does*.
- We've seen what a metacompiler *is*.

- C++ is a metacompiler.
- We've seen what a metacompiler *does*.
- We've seen what a metacompiler *is*.
- We now ask how a metacompiler *works*.

To be clear,

To be clear,

> We are still *one* section away from discussing the actual implementation.

To be clear,

> We are still *one* section away from discussing the actual implementation. This section offers an overview of the general methods that will be used.

Why talk about methods?

Why talk about methods?

A general purpose DSL engine needs to be able to metacompile any language,

Why talk about methods?

> A general purpose DSL engine needs to be able to metacompile any language, and so we need its implementation design to be based on expressive theoretical foundations.

We start with the methods of compiler theory, which is divided into the frontend and backend.

The frontend focuses on the lexing and parsing of source code.

*Intermediate Representation.

The frontend focuses on the lexing and parsing of source code.

The backend focuses on multilayed translations from an initial IR* assembly, to the final target assembly.

*Intermediate Representation.

Lexing

Lexing

To keep things simple, lexers:

Lexing

To keep things simple, lexers:

- Read source code.

Lexing

To keep things simple, lexers:

- Read source code.
- Translate words into tokens.

Lexing

To keep things simple, lexers:

- Read source code.
- Translate words into tokens.
- Are constructed from regular languages and regular automata.

Parsing

Parsing

To keep things simple, parsers:

Parsing

To keep things simple, parsers:

- Read tokenized source code.

Parsing

To keep things simple, parsers:

- Read tokenized source code.
- Confirm “sentence” structure.

Parsing

To keep things simple, parsers:

- Read tokenized source code.
- Confirm "sentence" structure.
- Translate sentences into IR assembly.

Parsing

To keep things simple, parsers:

- Read tokenized source code.
- Confirm “sentence” structure.
- Translate sentences into IR assembly.
- Are constructed from context free grammars and pushdown automata.

As for backend methods:

As for backend methods:

- Assembly languages are generally implemented using methods derived from register machines.

As for backend methods:

- Assembly languages are generally implemented using methods derived from register machines.
- Such methods coincide well with implementing imperative DSLs.

As for backend methods:

- Assembly languages are generally implemented using methods derived from register machines.
- Such methods coincide well with implementing imperative DSLs.
- Such methods are less effective when implementing functional DSLs.

We need methods that can implement both imperative and functional languages.

We need methods that can implement both imperative and functional languages.

To motivate such methods, let's now take a quick tour of computing history.

Turing machines:

Turing machines:

- Alan Turing, 1936.

Turing machines:

- Alan Turing, 1936.
- Equivalent to μ-recursive functions (math).

Turing machines:

- Alan Turing, 1936.
- Equivalent to μ-recursive functions (math).
- Well suited for modeling theoretical properties of computable functions.

Turing machines:

- Alan Turing, 1936.
- Equivalent to μ-recursive functions (math).
- Well suited for modeling theoretical properties of computable functions.
- Less well suited for modeling practical or performant computable functions.

Lambda calculus $[\lambda x.x]$:

Lambda calculus [$\lambda x.x$]:

- Alonzo Church, 1930s.

Lambda calculus [$\lambda x.x$]:

- Alonzo Church, 1930s.
- Equipotent to Turing machines.

Lambda calculus [$\lambda x.x$]:

- Alonzo Church, 1930s.
- Equipotent to Turing machines.
- Well suited for modeling theoretical grammar of computable functions.

Lambda calculus [$\lambda x.x$]:

- Alonzo Church, 1930s.
- Equipotent to Turing machines.
- Well suited for modeling theoretical grammar of computable functions.
- Less well suited (on its own) for modeling certain *consistency* semantics of computable functions.

LISP programming language:

LISP programming language:

- John McCarthy, late 1950s.

LISP programming language:

- John McCarthy, late 1950s.
- Influenced by the lambda calculus.

LISP programming language:

- John McCarthy, late 1950s.
- Influenced by the lambda calculus.
- Is now a family of languages, including Common Lisp, Scheme, Clojure, and Racket.

LISP programming language:

- John McCarthy, late 1950s.
- Influenced by the lambda calculus.
- Is now a family of languages, including Common Lisp, Scheme, Clojure, and Racket.
- Aligns well with the functional programming paradigm.

To delve further into functional programming, To delve further into functional programming, we need to equip the lambda calculus (or lisp) with a type theory.

To delve further into functional programming, we need to equip the lambda calculus (or lisp) with a type theory.

But first. . .

Set Theory [math]:

†See Russell's Paradox.

Set Theory [math]:

- Georg Cantor, late 1800s.

† See Russell's Paradox.

Set Theory [math]:

- Georg Cantor, late 1800s.
- A foundational language of mathematics.

†See Russell's Paradox.

Set Theory [math]:

- Georg Cantor, late 1800s.
- A foundational language of mathematics.
- Proof that there are "different sizes of infinity."

† See Russell's Paradox.

Set Theory [math]:

- Georg Cantor, late 1800s.
- A foundational language of mathematics.
- Proof that there are "different sizes of infinity."
- If taken as a naive theory, it leads to contradictions.[†]

[†]See Russell's Paradox.

Type Theory [math]:

Type Theory [math]:

- Bertrand Russell and Alfred North Whitehead, early 1900s.

Type Theory [math]:

- Bertrand Russell and Alfred North Whitehead, early 1900s.
- Principia Mathematica, intended as alternative to set theory.

Type Theory [math]:

- Bertrand Russell and Alfred North Whitehead, early 1900s.
- Principia Mathematica, intended as alternative to set theory.
- Mathematicians did not adopt this approach, instead vying for axiomatic set theory.

Type Theory [math]:

- Bertrand Russell and Alfred North Whitehead, early 1900s.
- Principia Mathematica, intended as alternative to set theory.
- Mathematicians did not adopt this approach, instead vying for axiomatic set theory.
- Helped advance the subject of symbolic logic.

Type Theory [computing]:

‡See Lambda Cube.

Type Theory [computing]:

- Multiple contributors (here unnamed), mid 1900s.

‡See Lambda Cube.

Type Theory [computing]:

- Multiple contributors (here unnamed), mid 1900s.
- More recently Per Martin-Löf, late 1900s.

‡See Lambda Cube.

Type Theory [computing]:

- Multiple contributors (here unnamed), mid 1900s.
- More recently Per Martin-Löf, late 1900s.
- Well suited for modeling certain *consistency* semantics of computable functions.

‡See Lambda Cube.

Type Theory [computing]:

- Multiple contributors (here unnamed), mid 1900s.
- More recently Per Martin-Löf, late 1900s.
- Well suited for modeling certain *consistency* semantics of computable functions.
- Aligns well with the lambda calculus[‡], functional programming, and the family of LISPs.

[‡]See Lambda Cube.

This leads us to. . .

This leads us to... abstract machines.

This leads us to... abstract machines.

We previously introduced this idea, but in the context of the lambda calculus and LISP, we can specifically mention SECD, CESK, and Krivine machines.

This leads us to... abstract machines.

We previously introduced this idea, but in the context of the lambda calculus and LISP, we can specifically mention SECD, CESK, and Krivine machines.

Each uses different grammatical artifacts from the untyped lambda calculus to implement its own version of an abstract machine.

Abstract machines

vs

Register machines

Abstract machines:

Abstract machines:

- Consist of some version of a controller, memory lookup, and call stack.

Abstract machines:

- Consist of some version of a controller, memory lookup, and call stack.
- They transition states by updating these components, which is how they perform their computations.

Register machines:

Register machines:

- They are abstract machines.

Register machines:

- They are abstract machines.
- The only difference is their design more closely resembles actual computer architecture.

Compile time register machines:[§]

[§]Practical TMP: A C++17 Compile Time Register Machine, C++Now 2021, Daniel Nikpayuk.

Compile time register machines:[§]

- Use continuation passing style to transition from state to state.

[§]Practical TMP: A C++17 Compile Time Register Machine, C++Now 2021, Daniel Nikpayuk.

Compile time register machines:[§]

- Use continuation passing style to transition from state to state.
- Use template parameters to carry their respective controllers, registers, and call stacks.

[§] Practical TMP: A C++17 Compile Time Register Machine, C++Now 2021, Daniel Nikpayuk.

Compile time register machines:[§]

- Use continuation passing style to transition from state to state.
- Use template parameters to carry their respective controllers, registers, and call stacks.
- Use nesting depth counters to implement trampolining as well as prevent recursive closure.

[§] Practical TMP: A C++17 Compile Time Register Machine, C++Now 2021, Daniel Nikpayuk.

Library

In this section we finally discuss how to implement a DSL engine in C++.

In this section we finally discuss how to implement a DSL engine in C++.

In order to do so I introduce my cctmp library.

cctmp library[¶]:

[¶]https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

cctmp library:

- An open source repository on GitHub.[¶]

[¶] https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

cctmp library:

- An open source repository on GitHub.¶
- Currently implemented under C++17.

¶https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

cctmp library:

- An open source repository on GitHub.[¶]
- Currently implemented under C++17.
- Does not use the standard library. (std::)

¶https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

cctmp library:

- An open source repository on GitHub.[¶]
- Currently implemented under C++17.
- Does not use the standard library. (std::)
- Last checked, roughly 20 000 lines of code.

[¶] https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

cctmp library:

- An open source repository on GitHub.[¶]
- Currently implemented under C++17.
- Does not use the standard library. (std::)
- Last checked, roughly 20 000 lines of code.
- Project status:

[¶] https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

cctmp library:

- An open source repository on GitHub.[¶]
- Currently implemented under C++17.
- Does not use the standard library. (std::)
- Last checked, roughly 20 000 lines of code.
- Project status: Currently proof of concept,

[¶] https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

cctmp library:

- An open source repository on GitHub.[¶]
- Currently implemented under C++17.
- Does not use the standard library. (std::)
- Last checked, roughly 20 000 lines of code.
- Project status: Currently proof of concept, but starting to mature and stabilize.

[¶] https://github.com/Daniel-Nikpayuk/cpp-cctmp-library

cctmp library, testing:

cctmp library, testing:

- Currently informal,

cctmp library, testing:

- Currently informal, otherwise:

cctmp library, testing:

- Currently informal, otherwise:
- Has been tested on several laptops.

cctmp library, testing:

- Currently informal, otherwise:
- Has been tested on several laptops.
- Has been tested using several versions of Ubuntu.

cctmp library, testing:

- Currently informal, otherwise:
- Has been tested on several laptops.
- Has been tested using several versions of Ubuntu.
- Compiles under several versions of GCC including 11.4.0.

cctmp library, testing:

- Currently informal, otherwise:
- Has been tested on several laptops.
- Has been tested using several versions of Ubuntu.
- Compiles under several versions of GCC including 11.4.0.
- Compiles under several versions of Clang including 14.0.0.

cctmp library, directories:

- 00 cctmp
- 01 assembly
- 02 generator
- 03 fileput
- 04 chord
- 05 hustle

cctmp library, directories:

- 00 cctmp - constexpr data structures, TMP magic.
- 01 assembly
- 02 generator
- 03 fileput
- 04 chord
- 05 hustle

cctmp library, directories:

- 00 cctmp - constexpr data structures, TMP magic.
- 01 assembly - Continuation machines, DSL engine.
- 02 generator
- 03 fileput
- 04 chord
- 05 hustle

cctmp library, directories:

- 00 cctmp - constexpr data structures, TMP magic.
- 01 assembly - Continuation machines, DSL engine.
- 02 generator - LL(1) parser generator.
- 03 fileput
- 04 chord
- 05 hustle

cctmp library, directories:

- 00 cctmp - constexpr data structures, TMP magic.
- 01 assembly - Continuation machines, DSL engine.
- 02 generator - LL(1) parser generator.
- 03 fileput - metaobject (controller) output.
- 04 chord
- 05 hustle

cctmp library, directories:

- 00 cctmp - constexpr data structures, TMP magic.
- 01 assembly - Continuation machines, DSL engine.
- 02 generator - LL(1) parser generator.
- 03 fileput - metaobject (controller) output.
- 04 chord - Lexer, parser, for the chord DSL.
- 05 hustle

cctmp library, directories:

- 00 cctmp - constexpr data structures, TMP magic.
- 01 assembly - Continuation machines, DSL engine.
- 02 generator - LL(1) parser generator.
- 03 fileput - metaobject (controller) output.
- 04 chord - Lexer, parser, for the chord DSL.
- 05 hustle - Lexer, parser, for the hustle DSL.

What about the metacompiler?

What about the metacompiler?

We begin with its frontend:

metacompiler frontend:

metacompiler frontend:

- The implementation is inspired by the “Dragon Book”.

metacompiler frontend:

- The implementation is inspired by the "Dragon Book".
- Currently each DSL handcodes its own lexer, based on automata theory.

metacompiler frontend:

- The implementation is inspired by the "Dragon Book".
- Currently each DSL handcodes its own lexer, based on automata theory.
- An LL(1) parser generator is used to construct transition tables for DSL context free grammars.

hustle language, context free grammar:

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

            "Start",

        // hustle:

            "Start   -> ( Generic )                    ;"
            "Generic -> type Param Params ( Main ) ;"
            "        -> Main                            ;"
            "Params  -> Param Params                    ;"
            "        -> empty                           ;"
            "Param   -> identifier : param_type        ;"

            // main:
```

hustle language, context free grammar:

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

         "Start",

        // hustle:

         "Start   -> ( Generic )                    ;"
         "Generic -> type Param Params ( Main )     ;"
         "        -> Main                           ;"
         "Params  -> Param Params                   ;"
         "        -> empty                          ;"
         "Param   -> identifier : param_type        ;"

          // main:
```

hustle language, context free grammar:

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

         "Start",

        // hustle:

         "Start   -> ( Generic )                    ;"
         "Generic -> type Param Params ( Main ) ;"
         "        -> Main                           ;"
         "Params  -> Param Params                   ;"
         "        -> empty                          ;"
         "Param   -> identifier : param_type        ;"

         // main:
```

hustle language, context free grammar:

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

            "Start",

        // hustle:

            "Start    -> ( Generic )                    ;"
            "Generic  -> type Param Params ( Main )     ;"
            "         -> Main                           ;"
            "Params   -> Param Params                   ;"
            "         -> empty                          ;"
            "Param    -> identifier : param_type        ;"

            // main:
```

hustle language, context free grammar:

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

           "Start",

        // hustle:

           "Start   -> ( Generic )                    ;"
           "Generic -> type Param Params ( Main )     ;"
           "        -> Main                           ;"
           "Params  -> Param Params                   ;"
           "        -> empty                          ;"
           "Param   -> identifier : param_type        ;"

           // main:
```

hustle language, context free grammar:

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

           "Start",

        // hustle:

           "Start   -> ( Generic )                    ;"
           "Generic -> type Param Params ( Main )     ;"
           "        -> Main                           ;"
           "Params  -> Param Params                   ;"
           "        -> empty                          ;"
           "Param   -> identifier : param_type        ;"

           // main:
```

hustle language, context free grammar:

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

            "Start",

        // hustle:

            "Start    -> ( Generic )                      ;"
            "Generic  -> type Param Params ( Main )       ;"
            "         -> Main                             ;"
            "Params   -> Param Params                     ;"
            "         -> empty                            ;"
            "Param    -> identifier : param_type          ;"

            // main:
```

hustle language, context free grammar:

```
constexpr auto source()
{
    return generator::context_free_grammar
    (
        // start:

           "Start",

        // hustle:

           "Start    -> ( Generic )                ;"
           "Generic  -> type Param Params ( Main ) ;"
           "         -> Main                       ;"
           "Params   -> Param Params               ;"
           "         -> empty                      ;"
           "Param    -> identifier : param_type    ;"

           // main:
```

metacompiler frontend [con't]:

metacompiler frontend [con't]:

- Syntax tree implementations inherit from a DSL engine base class.

metacompiler frontend [con't]:

- Syntax tree implementations inherit from a DSL engine base class.
- DSL source code translates into meta-assembly which acts as a controller for constructing constexpr functions.

As for meta-assembly:

As for meta-assembly:

How does such assembly translate into constexpr functions?

As for meta-assembly:

How does such assembly translate into constexpr functions?

We now turn our attention to the metacompiler backend:

metacompiler backend:

metacompiler backend:

- It is designed as an abstract machine, transitioning from state to state: $S_0 \rightarrow \ldots \rightarrow S_n$.

metacompiler backend:

- It is designed as an abstract machine, transitioning from state to state: $S_0 \to \ldots \to S_n$.
- Each state is a triple, consisting of:

metacompiler backend:

- It is designed as an abstract machine, transitioning from state to state: $S_0 \to \ldots \to S_n$.
- Each state is a triple, consisting of:
    - controller: A meta-assembly program which directs control flow.

metacompiler backend:

- It is designed as an abstract machine, transitioning from state to state: $S_0 \rightarrow \ldots \rightarrow S_n$.
- Each state is a triple, consisting of:
  - controller: A meta-assembly program which directs control flow.
  - universe: A variadic pack which acts as memory lookup.

metacompiler backend:

- It is designed as an abstract machine, transitioning from state to state: $S_0 \rightarrow \ldots \rightarrow S_n$.
- Each state is a triple, consisting of:
  - controller: A meta-assembly program which directs control flow.
  - universe: A variadic pack which acts as memory lookup.
  - stage: A variadic pack which acts as a call stack.

metacompiler backend:

- It is designed as an abstract machine, transitioning from state to state: $S_0 \to \ldots \to S_n$.
- Each state is a triple, consisting of:
  - controller: A meta-assembly program which directs control flow.
  - universe: A variadic pack which acts as memory lookup.
  - stage: A variadic pack which acts as a call stack.
- States are implemented using continuation passing style.

An example of
a continuation passing state:

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

continuation machine:

```
template<auto... filler>
struct T_assembly<name::go_to, note::id, filler...>
{
    template<assembly_params(c, i, l, t, r), typename... Ts>
    constexpr static auto result(Ts... vs)
    {
        constexpr auto ni = dispatch<c>::pos(i);

        return assembly_template(c, ni)
            ::assembly_result_ts(c, ni, l, t, r, Ts...)
                (vs...);
    }
};
```

Why continuation passing style?

Why continuation passing style?

If states are implemented as continuation passing functions,

Why continuation passing style?

If states are implemented as continuation passing functions, then meta-assembly not only controls state transitions,

Why continuation passing style?

If states are implemented as continuation passing functions, then meta-assembly not only controls state transitions, it also constructs a constexpr function along the way.

metacompiler backend (refinements):

‖ Structure and Interpretation of Computer Programs.

metacompiler backend (refinements):

- It is designed to simulate an abstract machine to achieve compilation.

‖ Structure and Interpretation of Computer Programs.

metacompiler backend (refinements):

- It is designed to simulate an abstract machine to achieve compilation.
- It constructs a sequence of state transitions, rather than applying them directly.

‖ Structure and Interpretation of Computer Programs.

metacompiler backend (refinements):

- It is designed to simulate an abstract machine to achieve compilation.
- It constructs a sequence of state transitions, rather than applying them directly.
- Although it is not an interpreter, its implementation is still inspired by SICP's‖ metacircular evaluator (for the Scheme language).

‖Structure and Interpretation of Computer Programs.

Before we go into the details of how the backend works,

Before we go into the details of how the backend works, it's worth discussing SICP's metacircular evaluator itself.

Before we go into the details of how the backend works, it's worth discussing SICP's metacircular evaluator itself.

It is implemented as an abstract machine with each state being a triple:

Before we go into the details of how the backend works, it's worth discussing SICP's metacircular evaluator itself.

It is implemented as an abstract machine with each state being a triple:

$$(\,\text{expression},\ \text{environment},\ \text{call stack}\,)$$

Details?

(expression, environment, call stack)

Details?

(expression, environment, call stack)

- expression: is the source code being translated and executed.
- environment: is a list of frames. Each frame is a list of bindings. A binding is a (variable, value) pair.
- call stack: is an expression constructed solely to apply a function to its respective values.

Details?

$$(\,\text{expression},\ \text{environment},\ \text{call stack}\,)$$

- expression: is the source code being translated and executed.
- environment: is a list of frames. Each frame is a list of bindings. A binding is a (variable, value) pair.
- call stack: is an expression constructed solely to apply a function to its respective values.

Details?

(expression, environment, call stack)

- expression: is the source code being translated and executed.
- environment: is a list of frames. Each frame is a list of bindings. A binding is a (variable, value) pair.
- call stack: is an expression constructed solely to apply a function to its respective values.

Details?

(expression, environment, call stack)

- expression: is the source code being translated and executed.
- environment: is a list of frames. Each frame is a list of bindings. A binding is a (variable, value) pair.
- call stack: is an expression constructed solely to apply a function to its respective values.

The crucial process for us to understand. . .

The crucial process for us to understand...

is how the call stack and environment interact when the source expression is being evaluated.

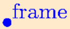
The crucial process for us to understand...

is how the call stack and environment interact when the source expression is being evaluated.

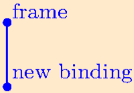
With that in mind, we have the following illustration:

environment

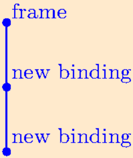
environment



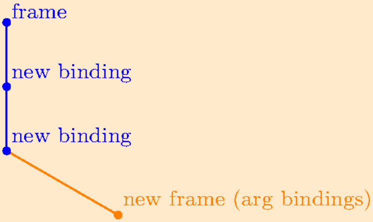
environment



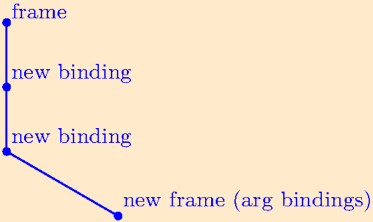
environment



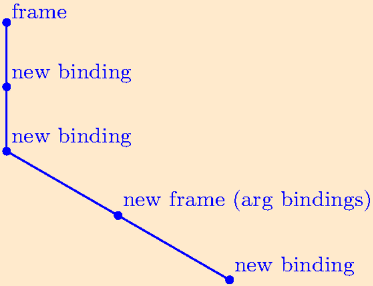
environment



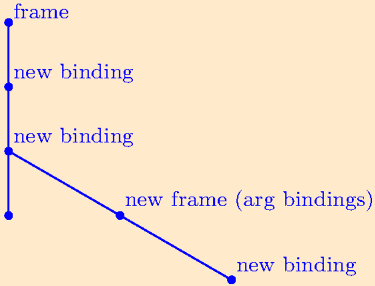
environment



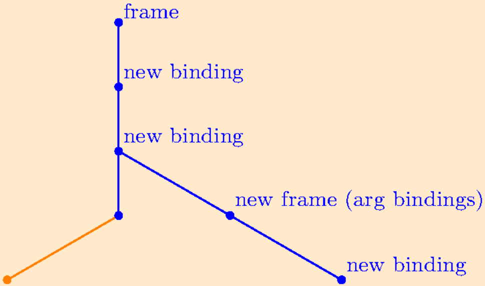
environment



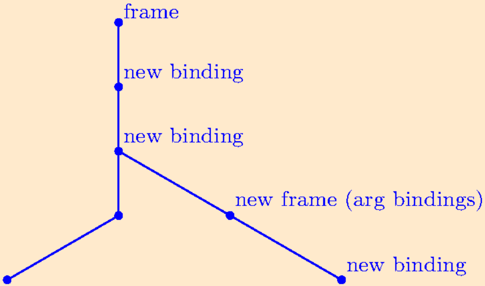
environment



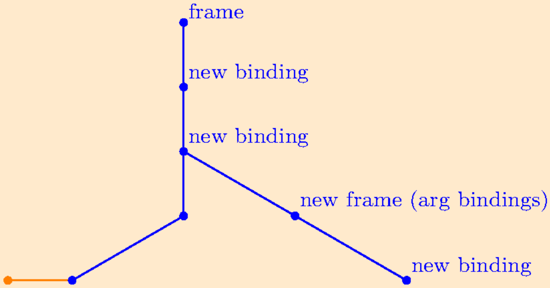
environment



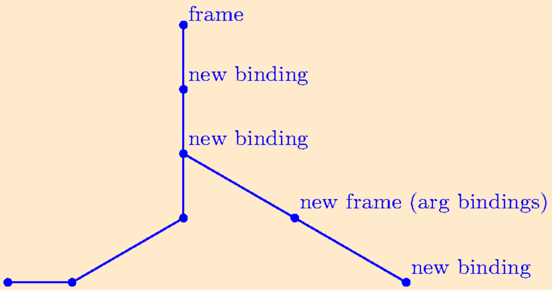
environment



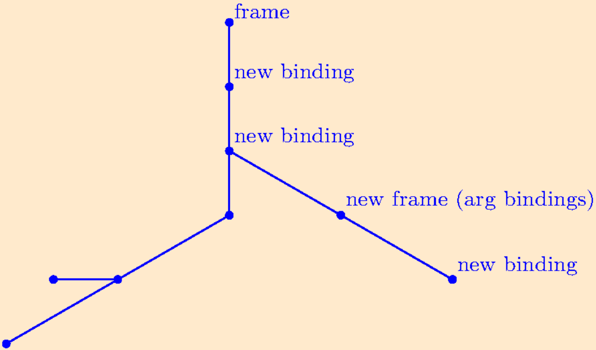
environment



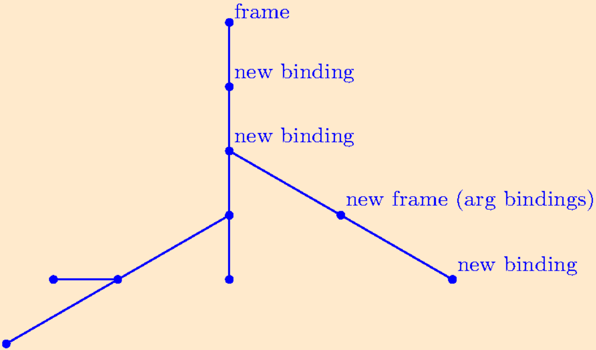
environment



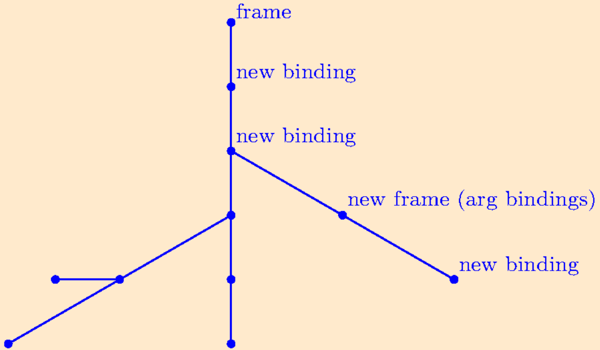
environment



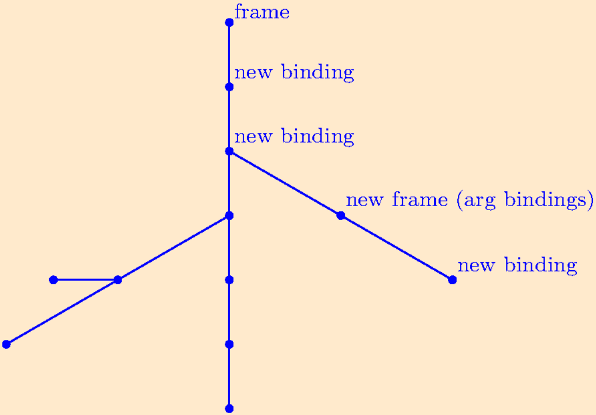
environment



environment



environment



Returning to our metacompiler abstract machine, let's review its components.

Returning to our metacompiler abstract machine, let's review its components.

Starting with the controller.

continuation machine controller:

continuation machine controller:

- It is a program composed of meta-assembly.

continuation machine controller:

- It is a program composed of meta-assembly.
- It is implemented as an array of instructions.

continuation machine controller:

- It is a program composed of meta-assembly.
- It is implemented as an array of instructions.
- Each instruction is an array of unsigned integers.

What does this assembly look like?

What does this assembly look like?

We saw it once before:

meta-assembly controller:

```
constexpr size_type value[][8] =
{
    { AN::id        , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::hash      , AT::port      ,  5,  0,  0,  0,  0,  1 },
    { AN::pad       , AT::select    ,  0,  1,  0,  0,  0,  1 },
    { AN::pad       , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::go_to     , AT::id        , 50,  0,  0,  0,  0,  1 },
    { AN::id        , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::eval      , AT::back      ,  7,  0,  0,  0,  0,  4 },
    { AN::id        , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::lookup    , AT::first     ,  0,  7,  0,  0,  0,  1 },
    { AN::halt      , AT::first     ,  0,  0,  0,  0,  0,  1 },
    { AN::eval      , AT::back      , 11,  0,  0,  0,  0,  5 },
    { AN::id        , AT::id        ,  0,  0,  0,  0,  0,  1 },
    { AN::arg       , AT::select    ,  1,  0,  0,  0,  0,  1 },
    { AN::arg       , AT::drop      ,  0,  0,  0,  0,  0,  1 },
    { AN::halt      , AT::first     ,  0,  0,  0,  0,  0,  1 },
    { AN::type      , AT::n_number  ,  0,  0,  0,  0,  0,  1 },
    { AN::literal   , AT::back      ,  0,  0,  0,  0,  0,  1 },
```

We now have enough to put some of these ideas together:

We now have enough to put some of these ideas together:

- The metacompiler frontend uses a SICP-style environment to keep track of variables.

We now have enough to put some of these ideas together:

- The metacompiler frontend uses a SICP-style environment to keep track of variables.
- Instead of keeping track of values, it holds an index (a promise) of where those values will eventually be in the continuation passing universe.

We now have enough to put some of these ideas together:

- The metacompiler frontend uses a SICP-style environment to keep track of variables.
- Instead of keeping track of values, it holds an index (a promise) of where those values will eventually be in the continuation passing universe.
- This is why the meta-assembly controller consists of numerical content only.

Next we discuss the universe:

continuation machine universe:

continuation machine universe:

- It is implemented using the left side of a variadic pack.

continuation machine universe:

- It is implemented using the left side of a variadic pack.
- New values are inserted at the end of the left side of the pack.

Finally, we have the stage:

continuation machine stage:

continuation machine stage:

- It is implemented using the right side of a variadic pack.

continuation machine stage:

- It is implemented using the right side of a variadic pack.
- New values are pushed to the back of the pack.

continuation machine stage:

- It is implemented using the right side of a variadic pack.
- New values are pushed to the back of the pack.
- Argument order is preserved when applying functions to their values.

Why use a variadic pack for both the universe and stage?

Why use a variadic pack for both the universe and stage?

To make it absolutely clear at this point, we are using the same pack ($Ts \ldots vs$) that is passed along the continuation machine states.

This works because of
a correspondence between:

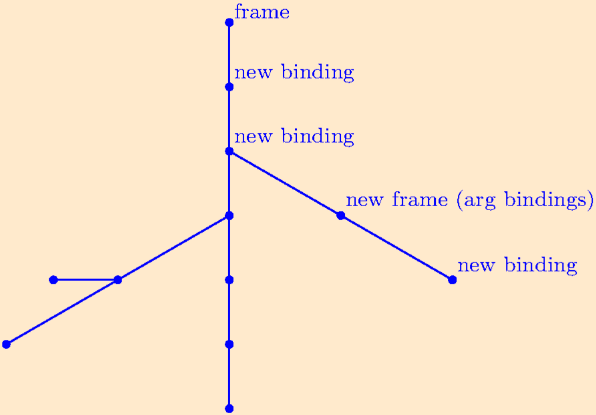
This works because of
a correspondence between:

- Environment variables (indices).

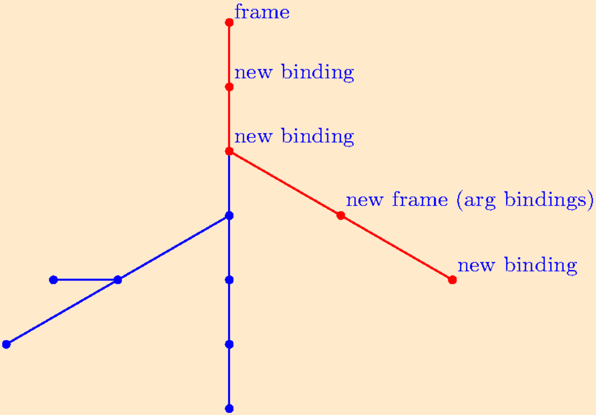
This works because of
a correspondence between:

- Environment variables (indices).
- The *flat* positions within a universe.

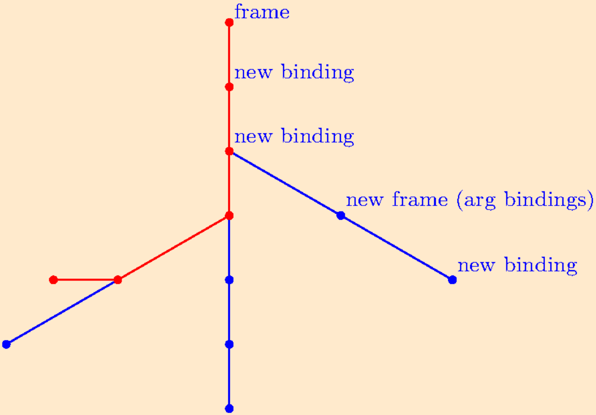
environment to universe correspondence:



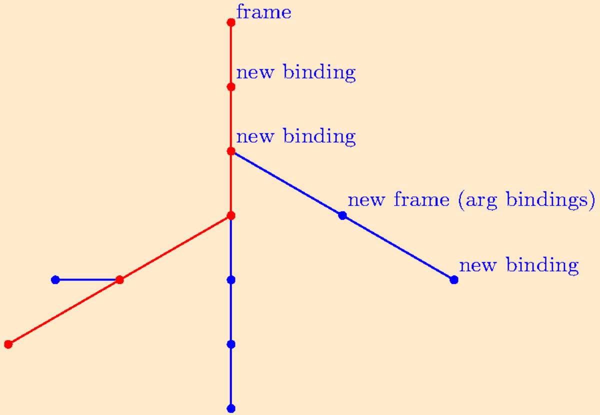
environment to universe correspondence:



environment to universe correspondence:



environment to universe correspondence:



Before we finish this section,

Before we finish this section,

We have one last technical issue to address,

Before we finish this section,
We have one last technical issue to address,
Which is of fundamental importance:

Before we finish this section,
We have one last technical issue to address,
Which is of fundamental importance:

Recursion.

How do you construct a recursive function from continuation passing style?

How do you construct a recursive function from continuation passing style?

The trick is to realize we only need call the same continuation machine with the same typed input.

How do you construct a recursive function from continuation passing style?

The trick is to realize we only need call the same continuation machine with the same typed input.

To do this in practice, we only need keep track of the initial context when defining the recursive function.

How do you construct a recursive function from continuation passing style?

The trick is to realize we only need call the same continuation machine with the same typed input.

To do this in practice, we only need keep track of the initial context when defining the recursive function.

We then supply that context before the arguments when applying said function.

This concludes how a metacompiler works in the context of this cctmp library.

Are there side effects in using continuation machines?

Are there side effects in using continuation machines?

- Type checking is deferred until a function is constructed.

Are there side effects in using continuation machines?

- Type checking is deferred until a function is constructed.
- This means that within DSL source code,

Are there side effects in using continuation machines?

- Type checking is deferred until a function is constructed.
- This means that within DSL source code, literals, variables, and function definitions don't require type info.

Are there side effects in using continuation machines?

- Type checking is deferred until a function is constructed.
- This means that within DSL source code, literals, variables, and function definitions don't require type info.
- Caveat: Recursive functions are the exception to this because C++ compilers don't like to *auto* deduce such things.

Are there side effects in using continuation machines?

- Type checking is deferred until a function is constructed.
- This means that within DSL source code, literals, variables, and function definitions don't require type info.
- Caveat: Recursive functions are the exception to this because C++ compilers don't like to *auto* deduce such things.
- Mutability semantics have an added level of semantic indirection.

As for recursive functions requiring type info?

As for recursive functions requiring type info?

It is easy enough to bake the necessary grammar into our DSLs:

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                 ;"
                "factorial n -> T       ;"

                "body:                  ;"
                "   test equal n 0      ;"
                "   branch done         ;"
                "   . = subtract n 1    ;"
                "   . = factorial _     ;"
                "   . = multiply n _    ;"
                "   return _            ;"

                "done:                  ;"
                "   return 1:T          ;"
        );
}
```

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "   test equal n 0     ;"
                "   branch done        ;"
                "   . = subtract n 1   ;"
                "   . = factorial _    ;"
                "   . = multiply n _   ;"
                "   return _           ;"

                "done:                 ;"
                "   return 1:T         ;"
        );
}
```

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                ;"
                "factorial n -> T      ;"

                "body:                 ;"
                "   test equal n 0     ;"
                "   branch done        ;"
                "   . = subtract n 1   ;"
                "   . = factorial _    ;"
                "   . = multiply n _   ;"
                "   return _           ;"

                "done:                 ;"
                "   return 1:T         ;"
        );
}
```

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
        return source
        (
                "type T                 ;"
                "factorial n -> T       ;"

                "body:                  ;"
                "   test equal n 0      ;"
                "   branch done         ;"
                "   . = subtract n 1    ;"
                "   . = factorial _     ;"
                "   . = multiply n _    ;"
                "   return _            ;"

                "done:                  ;"
                "   return 1:T          ;"
        );
}
```

chord language, factorial:

```
constexpr auto _chord_factorial_v0()
{
	return source
	(
		"type T              ;"
		"factorial n -> T    ;"

		"body:               ;"
		"  test equal n 0    ;"
		"  branch done       ;"
		"  . = subtract n 1  ;"
		"  . = factorial _   ;"
		"  . = multiply n _  ;"
		"  return _          ;"

		"done:               ;"
		"  return 1:T        ;"
	);
}
```

hustle language, factorial:

```
constexpr auto _hustle_factorial_v0()
{
    return source
    (
        "(type T                                    "
        "  (define (factorial n) -> T               "
        "    (if (= n 0)                            "
        "      1:T                                  "
        "      (* n (factorial (- n 1)))            "
        "    )                                      "
        "  )                                        "
        ")                                          "
    );
}
```

hustle language, factorial:

```
constexpr auto _hustle_factorial_v0()
{
    return source
    (
        "(type T                                 "
        "  (define (factorial n) -> T            "
        "    (if (= n 0)                         "
        "      1:T                               "
        "      (* n (factorial (- n 1)))         "
        "    )                                   "
        "  )                                     "
        ")                                       "
    );
}
```

Language

This section showcases an inventory of distinct grammars that I have introduced into my chord language.

This section showcases an inventory of distinct grammars that I have introduced into my chord language.

It presents some of the expressive potential in using this paradigm of embedded DSLs.

chord language, void effects (mutability):

```
constexpr auto _chord_void_effects_v0()
{
	return source
	(
		"main x y              ;"

		"body:                 ;"
		"   . = increment y    ;"
		"   void appoint !x _  ;"
		"   return x           ;"
	);
}
```

Here *appoint* is defined as:

$$\text{void appoint}(u, v)\ \{\ *u = v;\ \}$$

chord language, void effects (mutability):

```
constexpr auto _chord_void_effects_v0()
{
        return source
        (
                "main x y              ;"

                "body:                 ;"
                "   . = increment y    ;"
                "   void appoint !x _  ;"
                "   return x           ;"
        );
}
```

Here *appoint* is defined as:

$$\text{void appoint}(u, v)\ \{\ *u = v;\ \}$$

chord language, factorial (goto):

```
constexpr auto _chord_factorial_v3()
{
        return source
        (
                "main p n           ;"

                "loop:              ;"
                "test is_zero n     ;"
                "branch done        ;"
                "p = multiply p n ;"
                "n = decrement n    ;"
                "goto loop          ;"

                "done:              ;"
                "return p           ;"
        );
}
```

chord language, factorial (goto):

```
constexpr auto _chord_factorial_v3()
{
	return source
	(
		"main p n            ;"

		"loop:               ;"
		"test is_zero n      ;"
		"branch done         ;"
		"p = multiply p n    ;"
		"n = decrement n     ;"
		"goto loop           ;"

		"done:               ;"
		"return p            ;"
	);
}
```

chord language, factorial (goto):

```
constexpr auto _chord_factorial_v3()
{
        return source
        (
                "main p n              ;"

                "loop:                 ;"
                "test is_zero n        ;"
                "branch done           ;"
                "p = multiply p n      ;"
                "n = decrement n       ;"
                "goto loop             ;"

                "done:                 ;"
                "return p              ;"
        );
}
```

chord language, square (argument compose):

```
constexpr auto _chord_square_v1()
{
        return source
        (
                "main x                          ;"

                "vars:                           ;"
                "declare sq                      ;"

                "defs:                           ;"
                "sq # argpose[1]{multiply 0 0} ;"

                "body:                           ;"
                ". = sq x                        ;"
                "return _                        ;"
        );
}
```

chord language, square (argument compose):

```
constexpr auto _chord_square_v1()
{
        return source
        (
                "main x                              ;"

                "vars:                               ;"
                "declare sq                          ;"

                "defs:                               ;"
                "sq # argpose[1]{multiply 0 0}       ;"

                "body:                               ;"
                ". = sq x                            ;"
                "return _                            ;"
        );
}
```

chord language, square (argument compose):

```
constexpr auto _chord_square_v1()
{
	return source
	(
		"main x                              ;"

		"vars:                               ;"
		"declare sq                          ;"

		"defs:                               ;"
		"sq # argpose[1]{multiply 0 0}       ;"

		"body:                               ;"
		". = sq x                            ;"
		"return _                            ;"
	);
}
```

chord language, square (argument compose):

```
constexpr auto _chord_square_v1()
{
	return source
	(
		"main x                          ;"

		"vars:                           ;"
		"declare sq                      ;"

		"defs:                           ;"
		"sq # argpose[1]{multiply 0 0}   ;"

		"body:                           ;"
		". = sq x                        ;"
		"return _                        ;"
	);
}
```

chord language, square (argument compose):

```
constexpr auto _chord_square_v1()
{
	return source
	(
		"main x                              ;"

		"vars:                               ;"
		"declare sq                          ;"

		"defs:                               ;"
		"sq # argpose[1]{multiply 0 0}       ;"

		"body:                               ;"
		". = sq x                            ;"
		"return _                            ;"
	);
}
```

chord language, square (argument compose):

```
constexpr auto _chord_square_v1()
{
	return source
	(
		"main x                                  ;"

		"vars:                                   ;"
		"declare sq                              ;"

		"defs:                                   ;"
		"sq # argpose[1]{multiply 0 0}           ;"

		"body:                                   ;"
		". = sq x                                ;"
		"return _                                ;"
	);
}
```

chord language, sum of squares (subcompose):

```
constexpr auto _chord_sum_of_squares_v1()
{
    return source
    (
        "main x y                                    ;"

        "vars:                                       ;"
        "declare sq sum_of_sq                        ;"

        "defs:                                       ;"
        "sq        # argpose[1]{multiply 0 0}        ;"
        "sum_of_sq # subpose[2]{add sq sq}           ;"

        "body:                                       ;"
        ". = sum_of_sq x y                           ;"
        "return _                                    ;"
    );
}
```

chord language, sum of squares (subcompose):

```
constexpr auto _chord_sum_of_squares_v1()
{
    return source
    (
        "main x y                                       ;"

        "vars:                                          ;"
        "declare sq sum_of_sq                           ;"

        "defs:                                          ;"
        "sq        # argpose[1]{multiply 0 0}           ;"
        "sum_of_sq # subpose[2]{add sq sq}              ;"

        "body:                                          ;"
        ". = sum_of_sq x y                              ;"
        "return _                                       ;"
    );
}
```

chord language, sum of squares (subcompose):

```
constexpr auto _chord_sum_of_squares_v1()
{
    return source
    (
        "main x y                                  ;"

        "vars:                                     ;"
        "declare sq sum_of_sq                      ;"

        "defs:                                     ;"
        "sq         # argpose[1]{multiply 0 0}     ;"
        "sum_of_sq  # subpose[2]{add sq sq}        ;"

        "body:                                     ;"
        ". = sum_of_sq x y                         ;"
        "return _                                  ;"
    );
}
```

chord language, sum of squares (subcompose):

```
constexpr auto _chord_sum_of_squares_v1()
{
    return source
    (
        "main x y                                    ;"

        "vars:                                       ;"
        "declare sq sum_of_sq                        ;"

        "defs:                                       ;"
        "sq        # argpose[1]{multiply 0 0}        ;"
        "sum_of_sq # subpose[2]{add sq sq}           ;"

        "body:                                       ;"
        ". = sum_of_sq x y                           ;"
        "return _                                    ;"
    );
}
```

chord language, sum of squares (subcompose):

```
constexpr auto _chord_sum_of_squares_v1()
{
    return source
    (
        "main x y                                  ;"

        "vars:                                     ;"
        "declare sq sum_of_sq                      ;"

        "defs:                                     ;"
        "sq        # argpose[1]{multiply 0 0}      ;"
        "sum_of_sq # subpose[2]{add sq sq}         ;"

        "body:                                     ;"
        ". = sum_of_sq x y                         ;"
        "return _                                  ;"
    );
}
```

chord language, sum of squares (subcompose):

```
constexpr auto _chord_sum_of_squares_v1()
{
    return source
    (
        "main x y                                  ;"

        "vars:                                     ;"
        "declare sq sum_of_sq                      ;"

        "defs:                                     ;"
        "sq        # argpose[1]{multiply 0 0}      ;"
        "sum_of_sq # subpose[2]{add sq sq}         ;"

        "body:                                     ;"
        ". = sum_of_sq x y                         ;"
        "return _                                  ;"
    );
}
```

chord language, twice (currying):

```
constexpr auto _chord_twice_v0()
{
    return source
    (
        "main x                              ;"

        "vars:                               ;"
        "declare twice                       ;"

        "defs:                               ;"
        "twice # curry[1]{multiply two}      ;"

        "body:                               ;"
        ". = twice x                         ;"
        "return _                            ;"

        , binding("two", 2)
    );
}
```

chord language, twice (currying):

```
constexpr auto _chord_twice_v0()
{
    return source
    (
        "main x                                    ;"

        "vars:                                     ;"
        "declare twice                             ;"

        "defs:                                     ;"
        "twice # curry[1]{multiply two}            ;"

        "body:                                     ;"
        ". = twice x                               ;"
        "return _                                  ;"

        , binding("two", 2)
    );
}
```

chord language, map operator (functional programming):

```
constexpr auto _chord_array_square_v0()
{
    return source
    (
        "main out in end                          ;"

        "vars:                                    ;"
        "   declare arr_sq                        ;"

        "defs:                                    ;"
        "   arr_sq # map[1]{square||} [) [,)      ;"

        "body:                                    ;"
        "   . = arr_sq !out in end                ;"
        "   return _                              ;"

        , binding( "square" , _square_ )
    );
}
```

chord language, map operator (functional programming):

```
constexpr auto _chord_array_square_v0()
{
    return source
    (
        "main out in end                    ;"

        "vars:                              ;"
        "  declare arr_sq                   ;"

        "defs:                              ;"
        "  arr_sq # map[1]{square||} [) [,) ;"

        "body:                              ;"
        "  . = arr_sq !out in end           ;"
        "  return _                         ;"

        , binding( "square" , _square_ )
    );
}
```

chord language, map operator (functional programming):

```
constexpr auto _chord_array_square_v0()
{
    return source
    (
        "main out in end                       ;"

        "vars:                                 ;"
        "   declare arr_sq                     ;"

        "defs:                                 ;"
        "   arr_sq # map[1]{square||} [) [,)   ;"

        "body:                                 ;"
        "   . = arr_sq !out in end             ;"
        "   return _                           ;"

        , binding( "square" , _square_ )
    );
}
```

chord language, map operator (functional programming):

```
constexpr auto _chord_array_square_v0()
{
    return source
    (
        "main out in end                          ;"

        "vars:                                    ;"
        "   declare arr_sq                        ;"

        "defs:                                    ;"
        "   arr_sq # map[1]{square||} [) [,)      ;"

        "body:                                    ;"
        "   . = arr_sq !out in end                ;"
        "   return _                              ;"

        , binding( "square" , _square_ )
    );
}
```

chord language, map operator (functional programming):

```
constexpr auto _chord_array_square_v0()
{
    return source
    (
        "main out in end                        ;"

        "vars:                                  ;"
        "   declare arr_sq                      ;"

        "defs:                                  ;"
        "   arr_sq # map[1]{square||} [) [,)    ;"

        "body:                                  ;"
        "   . = arr_sq !out in end              ;"
        "   return _                            ;"

        , binding( "square" , _square_ )
    );
}
```

chord language, map operator (functional programming):

```
constexpr auto _chord_array_square_v0()
{
    return source
    (
        "main out in end                         ;"

        "vars:                                   ;"
        "   declare arr_sq                       ;"

        "defs:                                   ;"
        "   arr_sq # map[1]{square||} [) [,)     ;"

        "body:                                   ;"
        "   . = arr_sq !out in end               ;"
        "   return _                             ;"

        , binding( "square" , _square_ )
    );
}
```

chord language, map operator (functional programming):

```
constexpr auto _chord_array_square_v0()
{
    return source
    (
        "main out in end                    ;"

        "vars:                              ;"
        "  declare arr_sq                   ;"

        "defs:                              ;"
        "  arr_sq # map[1]{square||} [) [,) ;"

        "body:                              ;"
        "  . = arr_sq !out in end           ;"
        "  return _                         ;"

        , binding( "square" , _square_ )
    );
}
```

The interval symbols [) and [,) are options which communicate how the functional algorithm acts with regard to iterator endpoints.

The interval symbols [) and [,) are options which communicate how the functional algorithm acts with regard to iterator endpoints.

The options are defined as follows:

The interval symbols [) and [,) are options which communicate how the functional algorithm acts with regard to iterator endpoints.

The options are defined as follows:

- [ ] - `closed`
- ( ] - `opening`
- ( ) - `open`
- [ ) - `closing`

The interval symbols [) and [,) are options which communicate how the functional algorithm acts with regard to iterator endpoints.

The options are defined as follows:

- [ ] - `closed`: acts on the left and right endpoints.
- ( ] - `opening`
- ( ) - `open`
- [ ) - `closing`

The interval symbols [) and [,) are options which communicate how the functional algorithm acts with regard to iterator endpoints.

The options are defined as follows:

- [] - `closed`
- (] - `opening`: skips the left, acts on the right.
- () - `open`
- [) - `closing`

The interval symbols [) and [,) are options which communicate how the functional algorithm acts with regard to iterator endpoints.

The options are defined as follows:

- [ ] - `closed`
- ( ] - `opening`
- ( ) - `open`: skips the left, doesn't act on the right.
- [ ) - `closing`

The interval symbols [) and [,) are options which communicate how the functional algorithm acts with regard to iterator endpoints.

The options are defined as follows:

- [ ] - `closed`
- ( ] - `opening`
- ( ) - `open`
- [ ) - `closing`: acts on the left, not on the right.

chord language, vector addition (functional programming):

```
constexpr auto _chord_vector_add_v0()
{
    return source
    (
        "main out in end in1                    ;"

        "vars:                                  ;"
        "   declare vec_add                     ;"

        "defs:                                  ;"
        "   vec_add # map[2]{add||} [) [,) [)   ;"

        "body:                                  ;"
        "   . = vec_add !out in end in1         ;"
        "   return _                            ;"
    );
}
```

chord language, vector addition (functional programming):

```
constexpr auto _chord_vector_add_v0()
{
    return source
    (
        "main out in end in1                        ;"

        "vars:                                      ;"
        "   declare vec_add                         ;"

        "defs:                                      ;"
        "   vec_add # map[2]{add||} [) [,) [)       ;"

        "body:                                      ;"
        "   . = vec_add !out in end in1             ;"
        "   return _                                ;"
    );
}
```

chord language, vector addition (functional programming):

```
constexpr auto _chord_vector_add_v0()
{
    return source
    (
        "main out in end in1                        ;"

        "vars:                                      ;"
        "   declare vec_add                         ;"

        "defs:                                      ;"
        "   vec_add # map[2]{add||} [) [,) [)       ;"

        "body:                                      ;"
        "   . = vec_add !out in end in1             ;"
        "   return _                                ;"
    );
}
```

chord language, vector addition (functional programming):

```
constexpr auto _chord_vector_add_v0()
{
    return source
    (
        "main out in end in1                    ;"

        "vars:                                  ;"
        "   declare vec_add                     ;"

        "defs:                                  ;"
        "   vec_add # map[2]{add||} [) [,) [)   ;"

        "body:                                  ;"
        "   . = vec_add !out in end in1         ;"
        "   return _                            ;"
    );
}
```

chord language, vector addition (functional programming):

```
constexpr auto _chord_vector_add_v0()
{
    return source
    (
        "main out in end in1                                ;"

        "vars:                                              ;"
        "   declare vec_add                                 ;"

        "defs:                                              ;"
        "   vec_add # map[2]{add||} [) [,) [)               ;"

        "body:                                              ;"
        "   . = vec_add !out in end in1                     ;"
        "   return _                                        ;"
    );
}
```

chord language, vector addition (functional programming):

```
constexpr auto _chord_vector_add_v0()
{
    return source
    (
        "main out in end in1                        ;"

        "vars:                                      ;"
        "   declare vec_add                         ;"

        "defs:                                      ;"
        "   vec_add # map[2]{add||} [) [,) [)       ;"

        "body:                                      ;"
        "   . = vec_add !out in end in1             ;"
        "   return _                                ;"
    );
}
```

chord language, vector addition (functional programming):

```
constexpr auto _chord_vector_add_v0()
{
    return source
    (
        "main out in end in1                    ;"

        "vars:                                  ;"
        "   declare vec_add                     ;"

        "defs:                                  ;"
        "   vec_add # map[2]{add||} [) [,) [)   ;"

        "body:                                  ;"
        "   . = vec_add !out in end in1         ;"
        "   return _                            ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                             ;"

        "vars:                                       ;"
        "   declare sum                              ;"

        "defs:                                       ;"
        "   sum # fold[1]{add * @|@ @||} <> [,]      ;"

        "body:                                       ;"
        "   . = sum !out in end                      ;"
        "   return _                                 ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                                    ;"

        "vars:                                              ;"
        "   declare sum                                     ;"

        "defs:                                              ;"
        "   sum # fold[1]{add * @|@ @||} <> [,]              ;"

        "body:                                              ;"
        "   . = sum !out in end                             ;"
        "   return _                                        ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                                    ;"

        "vars:                                              ;"
        "   declare sum                                     ;"

        "defs:                                              ;"
        "   sum # fold[1]{add * @|@ @||} <> [,]             ;"

        "body:                                              ;"
        "   . = sum !out in end                             ;"
        "   return _                                        ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                              ;"

        "vars:                                        ;"
        "  declare sum                                ;"

        "defs:                                        ;"
        "  sum # fold[1]{add * @|@ @||} <> [,]        ;"

        "body:                                        ;"
        "  . = sum !out in end                        ;"
        "  return _                                   ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                                        ;"

        "vars:                                                  ;"
        "   declare sum                                         ;"

        "defs:                                                  ;"
        "   sum # fold[1]{add * @|@ @||} <> [,]                 ;"

        "body:                                                  ;"
        "   . = sum !out in end                                 ;"
        "   return _                                            ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                                  ;"

        "vars:                                            ;"
        "   declare sum                                   ;"

        "defs:                                            ;"
        "   sum # fold[1]{add * @|@ @||} <> [,]           ;"

        "body:                                            ;"
        "   . = sum !out in end                           ;"
        "   return _                                      ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                              ;"

        "vars:                                        ;"
        "   declare sum                               ;"

        "defs:                                        ;"
        "   sum # fold[1]{add * @|@ @||} <> [,]       ;"

        "body:                                        ;"
        "   . = sum !out in end                       ;"
        "   return _                                  ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                                  ;"

        "vars:                                            ;"
        "   declare sum                                   ;"

        "defs:                                            ;"
        "   sum # fold[1]{add * @|@ @||} <> [,]           ;"

        "body:                                            ;"
        "   . = sum !out in end                           ;"
        "   return _                                      ;"
    );
}
```

chord language, fold operator (functional programming):

```
constexpr auto _chord_sum_v0()
{
    return source
    (
        "main out in end                                    ;"

        "vars:                                              ;"
        "   declare sum                                     ;"

        "defs:                                              ;"
        "   sum # fold[1]{add * @|@ @||} <> [,]             ;"

        "body:                                              ;"
        "   . = sum !out in end                             ;"
        "   return _                                        ;"
    );
}
```

chord language, dot product (functional programming):

```
constexpr auto _chord_dot_product_v0()
{
    return source
    (
        "main out in end in1                                        ;"

        "vars:                                                      ;"
        "  declare dot_prod                                         ;"

        "defs:                                                      ;"
        "  dot_prod # fold[2]{add * @|multiply||} <> [,) [)         ;"

        "body:                                                      ;"
        "  . = dot_prod !out in end in1                             ;"
        "  return _                                                 ;"
    );
}
```

chord language, dot product (functional programming):

```
constexpr auto _chord_dot_product_v0()
{
    return source
    (
        "main out in end in1                                        ;"

        "vars:                                                      ;"
        "   declare dot_prod                                        ;"

        "defs:                                                      ;"
        "   dot_prod # fold[2]{add * @|multiply||} <> [,) [)        ;"

        "body:                                                      ;"
        "   . = dot_prod !out in end in1                            ;"
        "   return _                                                ;"
    );
}
```

chord language, convolution (functional programming):

```
constexpr auto _chord_convolution_v0()
{
    return source
    (
        "main out in end in1                                          ;"

        "vars:                                                        ;"
        "  declare conv                                               ;"

        "defs:                                                        ;"
        "  conv # fold[2]{add * @|multiply||} <> (-|+,] []            ;"

        "body:                                                        ;"
        "  . = conv !out end in in1                                   ;"
        "  return _                                                   ;"
    );
}
```

chord language, convolution (functional programming):

```
constexpr auto _chord_convolution_v0()
{
    return source
    (
        "main out in end in1                                        ;"

        "vars:                                                      ;"
        "   declare conv                                            ;"

        "defs:                                                      ;"
        "   conv # fold[2]{add * @|multiply||} <> (-|+,] []         ;"

        "body:                                                      ;"
        "   . = conv !out end in in1                                ;"
        "   return _                                                ;"
    );
}
```

Performance

Performance at this stage of development is informal.

Performance at this stage of development is informal.

With that said, I can still offer some basic stats.

C++17 language, square root (newton's method):

```
template<typename T>
constexpr auto sqrt_iter(T x, T guess) -> T
{
    auto tolerance          = 0.0001;
    auto square             = [](T y){ return y * y; };
    auto abs                = [](T y){ return (y < 0) ? -y : y; };
    auto good_enough        = [&](T g) { return (abs(square(g) - x) < tolerance); };

    auto average            = [](T y, T z){ return (y + z) / 2; };
    auto improve            = [&](T g){ return average(g, x/g); };

    if (good_enough(guess)) return guess;
    else                    return sqrt_iter(x, improve(guess));
}

template<typename T>
constexpr auto sqrt(T x) { return sqrt_iter<T>(x, 1.0); }
```

This isn't a DSL.

This isn't a DSL.

It is standard C++17.

This isn't a DSL.

It is standard C++17.

This is our baseline.

- Compile times:

- Compile times:
  - GCC - 0.100s
  - Clang - 0.114s

- Compile times:
  - GCC - 0.100s
  - Clang - 0.114s
- Run times:

- Compile times:
  - GCC - 0.100s
  - Clang - 0.114s
- Run times:
  - GCC - 0.001s
  - Clang - 0.001s

- Compile times:
  - GCC - 0.100s
  - Clang - 0.114s
- Run times:
  - GCC - 0.001s
  - Clang - 0.001s
- -01 binary size:

- Compile times:
  - GCC - 0.100s
  - Clang - 0.114s
- Run times:
  - GCC - 0.001s
  - Clang - 0.001s
- -01 binary size:
  - GCC - 16 176 B
  - Clang - 16 104 B

- Compile times:
  - GCC - 0.100s
  - Clang - 0.114s
- Run times:
  - GCC - 0.001s
  - Clang - 0.001s
- -01 binary size:
  - GCC - 16 176 B
  - Clang - 16 104 B
- -02/03 binary size:

- Compile times:
  - GCC - 0.100s
  - Clang - 0.114s
- Run times:
  - GCC - 0.001s
  - Clang - 0.001s
- -01 binary size:
  - GCC - 16 176 B
  - Clang - 16 104 B
- -02/03 binary size:
  - GCC - 16 128 B
  - Clang - 16 104 B

hustle language, square root (newton's method):

```
constexpr auto _hustle_square_root_v0()
{
  return source
  (
    "(type T                                                                  "
    "  (define (sqrt x)                                                       "

    "    (define (square y) (* y y))                                          "
    "    (define (abs y) (if (< y 0) (- y) y))                                "
    "    (define (good-enough? guess) (< (abs (- (square guess) x)) tolerance)) "
    "    (define (average y z) (/ (+ y z) 2))                                 "
    "    (define (improve guess) (average guess (/ x guess)) )                "
    "    (define (sqrt-iter guess) -> T                                       "
    "      (if (good-enough? guess) guess (sqrt-iter (improve guess)))        "
    "    )                                                                    "

    "    (sqrt-iter 1:T)                                                      "
    "  )                                                                      "
    ")                                                                        "

    , binding("tolerance", 0.0001)
  );
}
```

This is the same function implemented in hustle code.

This is the same function implemented in hustle code.

- Compile times:

This is the same function implemented in hustle code.

- Compile times:
  - GCC - 1.644s
  - Clang - 2.982s

This is the same function implemented in hustle code.

- Compile times:
  - GCC - 1.644s
  - Clang - 2.982s
- Run times:

This is the same function implemented in hustle code.

- Compile times:
  - GCC - 1.644s
  - Clang - 2.982s
- Run times:
  - GCC - 0.002s
  - Clang - 0.001s

This is the same function implemented in hustle code.

- Compile times:
  - GCC - 1.644s
  - Clang - 2.982s
- Run times:
  - GCC - 0.002s
  - Clang - 0.001s
- -01 binary size:

This is the same function implemented in hustle code.

- Compile times:
  - GCC - 1.644s
  - Clang - 2.982s
- Run times:
  - GCC - 0.002s
  - Clang - 0.001s
- -01 binary size:
  - GCC - 17 432 B
  - Clang - 16 624 B

This is the same function implemented in hustle code.

- Compile times:
  - GCC - 1.644s
  - Clang - 2.982s
- Run times:
  - GCC - 0.002s
  - Clang - 0.001s
- -01 binary size:
  - GCC - 17 432 B
  - Clang - 16 624 B
- -02/03 binary size:

This is the same function implemented in hustle code.

- Compile times:
  - GCC - 1.644s
  - Clang - 2.982s
- Run times:
  - GCC - 0.002s
  - Clang - 0.001s
- -01 binary size:
  - GCC - 17 432 B
  - Clang - 16 624 B
- -02/03 binary size:
  - GCC - 16 272 B
  - Clang - 16 632 B

Finally, I have a test suite of 20 chord functions.

Finally, I have a test suite of 20 chord functions.

Metacompiling these all at once we get:

Finally, I have a test suite of 20 chord functions.
Metacompiling these all at once we get:

- Compile times:

Finally, I have a test suite of 20 chord functions. Metacompiling these all at once we get:

- Compile times:
  - GCC - 4.108s
  - Clang - 9.004s

Finally, I have a test suite of 20 chord functions.

Metacompiling these all at once we get:

- Compile times:
  - GCC - 4.108s
  - Clang - 9.004s
- Run times:

Finally, I have a test suite of 20 chord functions.

Metacompiling these all at once we get:

- Compile times:
  - GCC - 4.108s
  - Clang - 9.004s
- Run times:
  - GCC - 0.002s
  - Clang - 0.002s

Finally, I have a test suite of 20 chord functions.

Metacompiling these all at once we get:

- Compile times:
  - GCC - 4.108s
  - Clang - 9.004s
- Run times:
  - GCC - 0.002s
  - Clang - 0.002s
- -01 binary size:

Finally, I have a test suite of 20 chord functions.

Metacompiling these all at once we get:

- Compile times:
  - GCC - 4.108s
  - Clang - 9.004s
- Run times:
  - GCC - 0.002s
  - Clang - 0.002s
- -01 binary size:
  - GCC - 19 320 B
  - Clang - 18 152 B

Finally, I have a test suite of 20 chord functions.

Metacompiling these all at once we get:

- Compile times:
  - GCC - 4.108s
  - Clang - 9.004s
- Run times:
  - GCC - 0.002s
  - Clang - 0.002s
- -01 binary size:
  - GCC - 19 320 B
  - Clang - 18 152 B
- -02/03 binary size:

Finally, I have a test suite of 20 chord functions.

Metacompiling these all at once we get:

- Compile times:
  - GCC - 4.108s
  - Clang - 9.004s
- Run times:
  - GCC - 0.002s
  - Clang - 0.002s
- -01 binary size:
  - GCC - 19 320 B
  - Clang - 18 152 B
- -02/03 binary size:
  - GCC - 17 896 B / 17 896 B
  - Clang - 17 320 B / 16 568 B

Roadmap

As previously stated, this project and its library are still proof of concept.

As previously stated, this project and its library are still proof of concept.

As such, and to wind down this talk, I offer a roadmap for future directions.

The thing to note is that for as much as this paradigm is stabilizing, there are still open areas of research.

The thing to note is that for as much as this paradigm is stabilizing, there are still open areas of research.

I'm willing to say the main reason for this is because of the self-similarity of these designs.

This self-similarity creates what I would call a two-tier system.

This self-similarity creates what I would call a two-tier system.

The first tier is C++ itself. It has a first tier type system, and mutability semantics, etc.

This self-similarity creates what I would call a two-tier system.

The first tier is C++ itself. It has a first tier type system, and mutability semantics, etc.

The second tier is any given DSL, which can potentially have their own type systems, mutability semantics, etc.

For example the C++ compiler offers error messages, but what about the DSLs?

For example the C++ compiler offers error messages, but what about the DSLs?

Currently they do not,

For example the C++ compiler offers error messages, but what about the DSLs?

Currently they do not, but they will.

For example the C++ compiler offers error messages, but what about the DSLs?

Currently they do not, but they will.

What's more, you can design for their inclusion the same way you would for any compiler.

For example the C++ compiler offers error messages, but what about the DSLs?

Currently they do not, but they will.

What's more, you can design for their inclusion the same way you would for any compiler.

Also, as there's very little TMP magic in this library, and given that continuation machines have a *step-by-step* nature,

For example the C++ compiler offers error messages, but what about the DSLs?

Currently they do not, but they will.

What's more, you can design for their inclusion the same way you would for any compiler.

Also, as there's very little TMP magic in this library, and given that continuation machines have a *step-by-step* nature, this means that errors at their level are relatively easy to track.

In anycase, I want to say this two-tier system has some benefits in its design.

In anycase, I want to say this two-tier system has some benefits in its design.

Notably, it mitigates the learning curve for anyone exploring this paradigm.

In anycase, I want to say this two-tier system has some benefits in its design.

Notably, it mitigates the learning curve for anyone exploring this paradigm.

For example, there may be some new details, but if you already know how assembly works then meta-assembly is largely the same.

With that said, everything has a tradeoff, and there are some disadvantages as well.

With that said, everything has a tradeoff, and there are some disadvantages as well.

In the realm of context-switching, names matter. This is to prevent confusion and general misunderstanding.

With that said, everything has a tradeoff, and there are some disadvantages as well.

In the realm of context-switching, names matter. This is to prevent confusion and general misunderstanding.

To use assembly as example again, if you say the word “assembly” do you mean hardware or continuation machine?

As for open areas of research, I offer the two-tier type system as example:

The two-tier type system:

The two-tier type system:

- Metacompiled DSLs are deferentially typed.

The two-tier type system:

- Metacompiled DSLs are deferentially typed.
  - They are statically typed, but defer type checking until the constexpr function is built.

The two-tier type system:

- Metacompiled DSLs are deferentially typed.
  - They are statically typed, but defer type checking until the constexpr function is built.
  - What if we want our DSL to have a complete type system of its own?

The two-tier type system:

- Metacompiled DSLs are deferentially typed.
  - They are statically typed, but defer type checking until the constexpr function is built.
  - What if we want our DSL to have a complete type system of its own?
  - How best to negotiate between the tiers?

The two-tier type system [con't]:

The two-tier type system [con't]:

- Continuation machines potentially introduce DSL undefined behaviour (UB).

The two-tier type system [con't]:

- Continuation machines potentially introduce DSL undefined behaviour (UB).
  - DSL variables are untyped, and so you can assign new values of different types to these variables.

The two-tier type system [con't]:

- Continuation machines potentially introduce DSL undefined behaviour (UB).
  - DSL variables are untyped, and so you can assign new values of different types to these variables.
  - What safeguards should be set, and how best to do it?

The two-tier type system [con't]:

The two-tier type system [con't]:

- This paradigm introduces the possibility of reifying meta-assembly.

The two-tier type system [con't]:

- This paradigm introduces the possibility of reifying meta-assembly.
  - Do we create a second tier of data structures specific to DSL languages?

The two-tier type system [con't]:

- This paradigm introduces the possibility of reifying meta-assembly.
  - Do we create a second tier of data structures specific to DSL languages?
  - How should they interoperate with compile time C++ more generally?

As for the roadmap, the first step is to finish implementing the hustle language.

As for the roadmap, the first step is to finish implementing the hustle language.

For example, the following code does not yet work:

```
constexpr auto _hustle_conditional_operator_v0()
{
	return source
	(
		"(define (main n)          "
		"   ((if (= n 0) + *) 2 3) "
		")                         "
	);
}
```

More generally, the roadmap is as follows:

More generally, the roadmap is as follows:

- Use the hustle and chord DSLs to reimplement the cctmp library to become semiself hosting.

More generally, the roadmap is as follows:

- Use the hustle and chord DSLs to reimplement the cctmp library to become semiself hosting.
- Use this semiself hosting library to prototype and recreate better versions of itself.

More generally, the roadmap is as follows:

- Use the hustle and chord DSLs to reimplement the cctmp library to become semiself hosting.
- Use this semiself hosting library to prototype and recreate better versions of itself.
- Extend this library to include:

More generally, the roadmap is as follows:

- Use the hustle and chord DSLs to reimplement the cctmp library to become semiself hosting.
- Use this semiself hosting library to prototype and recreate better versions of itself.
- Extend this library to include:
  - A lexer generator.

More generally, the roadmap is as follows:

- Use the hustle and chord DSLs to reimplement the cctmp library to become semiself hosting.
- Use this semiself hosting library to prototype and recreate better versions of itself.
- Extend this library to include:
    - A lexer generator.
    - A LR(1) parser generator.

More generally, the roadmap is as follows:

- Use the hustle and chord DSLs to reimplement the cctmp library to become semiself hosting.
- Use this semiself hosting library to prototype and recreate better versions of itself.
- Extend this library to include:
  - A lexer generator.
  - A LR(1) parser generator.
  - A data structure generator.

More generally, the roadmap is as follows:

- Use the hustle and chord DSLs to reimplement the cctmp library to become semiself hosting.
- Use this semiself hosting library to prototype and recreate better versions of itself.
- Extend this library to include:
  - A lexer generator.
  - A LR(1) parser generator.
  - A data structure generator.
- Use this extended library to research and resolve open problems.

Finally, I will add. . .

Finally, I will add...

I have no ideal timeline for resolving these open problems, but I'm *hoping* to code the semiself hosting library, with extensions, within a year.

End

(thank you)

Questions?