Grammatical Elements of Function Induction

Daniel Nikpayuk

May 8, 2020

(updated August 26, 2020)*

Abstract

There is a correspondence between function construction grammar and finite state automata.

It is my intention in this essay to demonstrate the existence of this correspondence and its variations. Such correspondences are of theoretical importance as they provide bases toward type theoretic induction operators for function types. They are also of practical importance as they offer ways to create inventories of variant or performant functions without having to code each function manually. It is also my intention then to provide a semi-formal walk-through in how to build functions that build functions, as well as the grammars that allow for such possibilities.

Philosophy

The underlying design put forth in this essay—the one which informs all other designs here—is the idea of a correspondence, in particular between *function kinds* and the finite state *automata* otherwise equivalent to the Chomsky hierarchy of formal grammars. This design not only informs our ability to classify functions, it can also be used as a basis of translation for type inductive grammars. As there's a lot to unpack here, let's go over this.

First, a bit of background as to the meaning of the phrase *function induction*: In modern type theories such as [1] there's a special function for each *type* called the **induction** function, which is also called *dependent elimination*. As a rule, it effectively tells us what kind of functions are even possible for its given *type* definition, but we can also think of it as the function used to build other functions acting on the type. The philosophy of this essay takes inspiration from this formal version of induction and generalizes it as a concept to include broader classes of higher order functions, themselves defining classes of functions.

As for the correspondence, we'll get into the *function kinds* and *automata* details below, but the intuition inspiring this particular design is the idea that an **interpretation** acts as the common thread between evaluating a function and constructing that same function. Informally, we can say the only major difference between evaluation and construction here is that we *read* the interpretation when evaluating, while we *write* the interpretation when constructing.

*Updates are to fix the incorrect classification of certain operators. In particular `stem`, `costem`, and `distem` "bind" operators have now been reclassified as *conditional endopose* operators. The essay itself has been extended to clarify and refine the necessary explanations. Some minor grammatical fixes have also been made.

This may or may not be a novel idea for the reader (it was for me), but at the very least I feel it's not controversial to accept half of the claim more formally stated as:

If we can express how a function is evaluated, we can express how it is constructed

On the other hand, its converse:

If we can express how a function is constructed, we can express how it is evaluated

is not always true. The major claim of this essay restated another way then is that there are significant contexts in which this converse does hold.

Methodology

Based on the above philosophy, the methodology here can be split into two major perspectives: Evaluation, and construction. With that said, there is one additional perspective we will consider first: Prerequisites.

Prerequisites

The idea of our prerequisites are that for whatever grammars we do eventually come up with, we will want them to satisify certain design specifications (constraints) regardless of our other narratives. We begin with notation.

Function Notation

In terms of algorithmic analysis in this essay I will use a combination of notations from the LISP programming language, and what I call **grammatical path theory** which is independent research I've written up in [2]. Given this, any grammar we come up with should be amenable to these notations as well. With that said, there is no formal requirement for the casual reader to know these prerequisites, and for that reason I have largely kept both LISP as well as grammatical path explanations at an intuitive level—providing basic explanations when needed.

Modelling Strategies

For whatever grammars we do come up with as induction operators, we will need to be able to model their functions before we can specialize them. An alternative way to view such a task is to consider all such functions together as a *space*. How then would we go about constructing and representing such a space's elements?

To do this we will want our grammars to possess a **nonlinear combination** design. To explain this, we first appeal to the idea of a *vector space* which uses the concept of a *linear combination*:

$$a_1\,\mathbf{v}_1 \;+\; \dots \;+\; a_n\,\mathbf{v}_n$$

This represents all vectors in the *span* of a given basis. Notably such a representation is considered to be a **closed form**, or in other words a **model**.[1] We now deconstruct the idea of a linear combination to be applicable to broader spaces by considering two canonical approaches to *modelling*: Constructive (emergent) combinations which we'll here call *primary* models, as well as constrictive submodels of those primaries, which we'll call *secondary* models. In such a paradigm linear combinations are primary models.

As for an example of a secondary model, we could implement one by means of a primary such as $\mathbb{R}^2$, with which we can subset to derive the unit closed disk:

$$\mathcal{D} \;:=\; \{\ (x,y) \in \mathbb{R}^2 \mid x^2 + y^2 \le 1\ \}$$

With this said, both modelling strategies have their strengths and weaknesses and are each suited for different contexts. For example, the secondary (submodelling) approach is better suited for function design. It's one thing to build a function, but we have to know what to build in the first place—which often means

[1]One could even say the term "closed form" is the syntactic analog to the semantic "model".

we need to design then translate that into a construction. Submodelling thus allows us to both construct and to conceptualize top-down what's needed to eventually build things bottom-up.[2]

As for the primary emergent combination approach: Due to the desire for performance it is usually a better fit when actually implementing our function construction since then nothing is wasted.[3] Finally, this is where the *nonlinear combination* phrase comes from—to emphasize its use in function construction as similar to the above linear combinations.

To reiterate: Functions will be modelled as combinations by means of composition, and we take the name *nonlinear* because such functions will be shown to be built exactly in that way—nonlinearly.

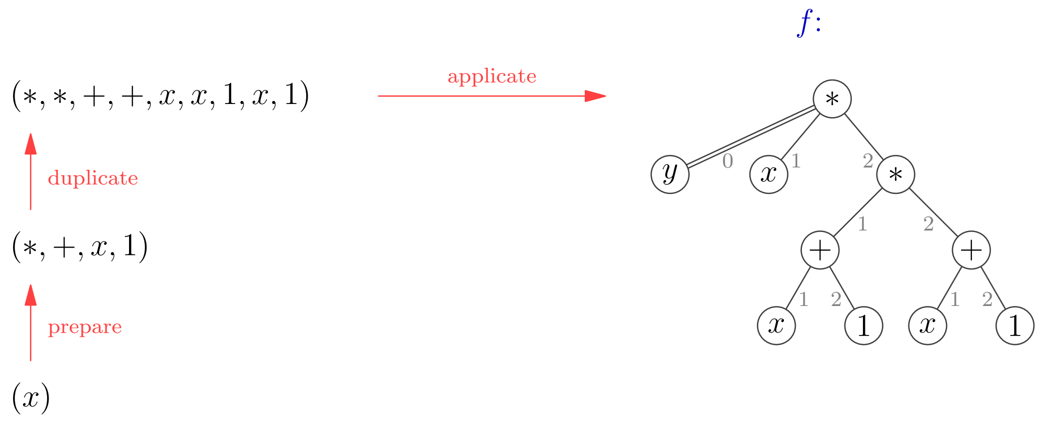
Compression

This is very much a practical design specification.

In application, nonlinear combinations allow us to define functions by breaking them up into smaller more manageable functions, but in doing so we actually create a new problem for ourselves: **Compression**, which plays a more active role in function definitions than we might realize.

In fact we compress for two clear reasons: The first is just to make our human user notations easier to read and work with. The following function has a sufficiently complicated grammatical path representation such that we would want to refactor its information and hide any of the details we intended to take for granted:

$$y = f(x) = x(x+1)^2$$



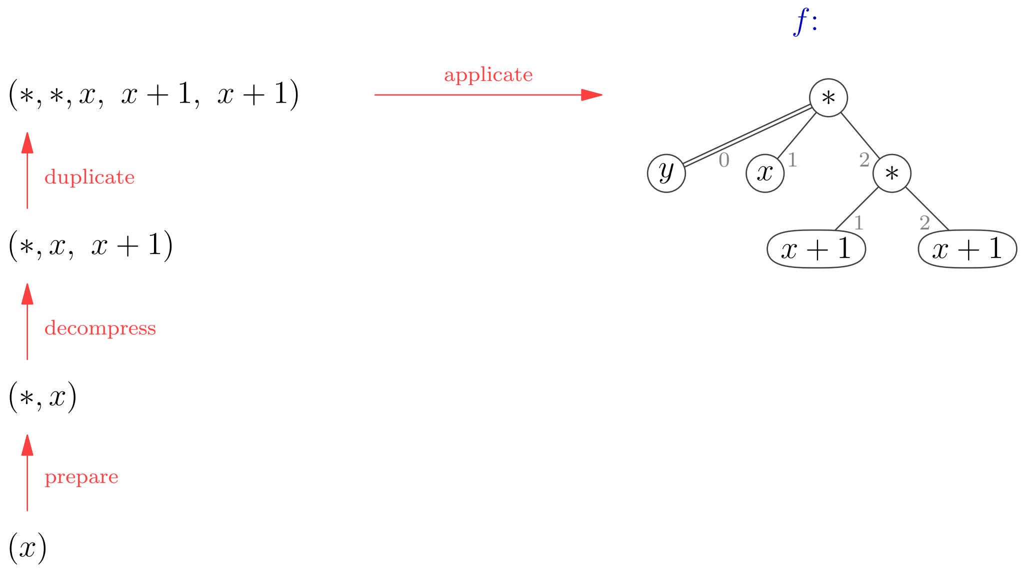
I describe this diagram as *canonical*, doing so with the sense that the tree component of this grammatical path notation is of maximum size. Regardless of this canonicity, if we go in the reverse direction of the mapping arrows we're able to compress the function signature—first reducing redundancy, then hiding those values that are otherwise held constant—allowing us to simplify all the way down to (x).

The second reason we compress is for computational performance. If you'll notice in the above diagram there is some redundancy with the $(x+1)$ expressions, which if we calculated separately would be wasteful. In this case, if we were to again refactor the tree into a signature, we could take advantage of this repetition and simplify:

[2]I should mention that using submodelling for design is possibly my own bias, but I find having a single umbrella function as a model in a context really does help clarify the components of design that are used to translate more refined models.

[3]The other advantage to taking this approach is that such combinations create a coded combinatorial inventory: We know exactly what is suppose to be in this space, and this has both theoretical and contextual applications. For example it allows us to make room for its functions when we're organizing a code library. We can then take a lazy policy where we would only be required to build the functions we need when we need. Later down the road we would then have a location for them for when it's time to add them in.

$$y = f(x) = x(x+1)^2$$



So far so good, but how then does compression become an issue here?

Compression works best with **complete information**. Generally, we would comprehend the bigger picture (all at once) to find those repetitive patterns we'd want to compress. This becomes an issue when we *divide and express* our above mentioned functions through nonlinear combinations: By modularizing out simpler functions we create information silos—we isolate our ability to recognize redundant patterns across subfunctions. We can still compress each component function, but our ability to compress overall along with our compression ratios will on average lessen. Can we mitigate this issue?

Yes, fortunately. We mitigate this issue by realizing that the **applicate** function above—which bijectively maps the *raw* (uncompressed) signature content to the tree nodes—can be reinterpreted to handle **just in time** compression: We equip our applicate function with the idea of "lazy evaluation" creating a kind of "delayed applicate". It has the same mapping pattern as before, but now it only maps as needed.

The implication of this change in perspective is we can now map parts of the signature to the whole tree piece by piece—parts corresponding to the component functions used to define the function whole. In this sense we have what I would call a **scope signature**, a component of local memory that spans across each module function. In practice what this means is we can *after the fact* push our compressions—otherwise redundant computations—onto this scope memory to be used across the range of function components.

Any grammar we then come up with for our induction operators should be able to simulate such scope signatures.

Signatures

One of the important ways to create variations of functions is to create variations within their signatures. The act of creating these variations is known as **the signature problem**.

Once the best practice solutions are known this problem is rather straightforward, but up until then it can be considered complicated and as we'll come to know is in fact what prevents us from initially creating more powerful versions of our intended construction grammar. It is for this reason that here I've decided to present a quick review of **tuples** and how they're built and used to implement function signatures. Later in the essay once a baseline of grammar is introduced we will then deal with the signature problem itself.

As for tuples, one can consider such objects to each have a given *length*, and are otherwise defined by the independent construct known as a **pair**:

$$(a, b, c, \ldots, z) = (a, (b, (c, (\ldots, (z,\ \text{null}\,) \ldots))))$$

Tuples then are recursive nestings of pairs with their final value always being the 0-tuple, also known as the *null* tuple.

Beyond this basic definition there are tuple operators, for which we now take an inventory of their primitives. In fact, there are four that will be shown as sufficient to model the rest:

$$\{\,\text{cons},\ \text{car},\ \text{cdr},\ \text{eq?}\,\}$$

The first three are defined as follows:

$$\begin{aligned} \text{cons}(x,y) &:= (x,y) \\ \text{car}((x,y)) &:= x \\ \text{cdr}((x,y)) &:= y \end{aligned}$$

The definition of the fourth (`eq?`) is slightly trickier as it requires recursion, so we'll take its definition for granted. The **cons**, **car**, **cdr** (and **eq?**) terminology I borrow from the LISP programming language. In particular, `cons` is the pair constructor, `car` is the first element projection function, and `cdr` the second element projection function. In coding literature the second element of a tuple is frequently referred to as the *rest* (of the tuple following the first element), which is a convention we will use in the following.

There are actually three *auxiliary* functions we'll want as well:

$$\{\,\text{push},\ \text{reverse},\ \text{concat}\,\}$$

The `push` function *appends* a value to the back of a tuple, analogous to how `cons` *prepends* one to the beginning. Versions of this function which are considered performant require more complicated implementations of tuples, as such we can't always assume users have this access with respect to their given contexts. In that case we will at least need `reverse` which takes a tuple as input and returns the tuple with the same content but in (componentwise) reversed order. Finally, `concat` takes two tuples and concatenates them together as one.

Before continuing with further methodology, I here implement a basic "length" function in LISP style grammar with C style comments in order to demonstrate how the initial primitives mentioned above can be used:[4]

```
(define (length′ count tuple)                          // declare and define length′
        (if (eq? tuple null)                           // if tuple == null
            count                                      // return count
           (length′ (+ count 1) (cdr tuple))           // else (recursively) call
        )                                              // (length′ (count + 1) rest)
)
```

By the way, if the above version of the tuple `length` function is unfamiliar, we can achieve the more intuitive interpretation as a specialization:

$$\text{length}(\textit{tuple}) \ := \ \text{length}'(0,\ \textit{tuple})$$

Evaluation

The major claim in this essay is of the correspondence between function construction grammar and finite state automata. If that's the case, how do automata even come into the picture in the first place?

Finite state automata in particular come into play here because the theory of automata espouses the idea that function **evaluation** corresponds to *membership tests* for strings within languages. If there really is a correspondence between function construction and evaluation, then by transitivity there should also be one between construction and automata.

The advantage of us exploring this connection by using automata is that automata theory already has well developed ideas of function evaluation, as well as logical expressivity (membership testing) which we can use to prove theorems about the induction grammar we eventually decide upon. As such, it would be best if the reader already has some familiarity with automata theory, but for a smoother transition I will provide a quick review here, as well as more specific details later on regarding each variety when we come across them.

[4]In fact, `eq?`, `reverse`, `push`, `concat` can all be implemented in similar ways.

Finite State Automata

The review I offer here is largely a combination of elements from [3] and [4].

To begin, automata is known to be equivalent to the *Chomsky hierarchy of formal grammars* which I find provides a simpler narrative for this introduction, and is the basis for the correspondence between evaluation and membership:

1. **Regular Languages** - correspond to regular automata as well as regular expressions.
2. **Context-Free Languages** - correspond to pushdown automata.
3. **Context-Sensitive Languages** - correspond to linear bounded automata.
4. **Semidecidable Languages** - correspond to register machines.[5]

You can think of each level in the hierarchy as being built on top of the previous levels, where regular languages are simplest and semideciable languages are most complex.

As for the definition of a **language**, we start with an *alphabet* Σ, and create *strings* of a given length Σ^n (for arbitrary lengths n) and then we take the set of all strings to be:

$$\Sigma^* \;:=\; \bigcup_{n\geq 0} \Sigma^n$$

where $\Sigma^0 := \{\epsilon\}$, the set containing the empty string. A language is defined to be a secondary model derived from this primary:

$$L \subseteq \Sigma^*$$

As for the correspondence between languages and automata, it comes from the fact that *finite state automata* are able to recognize whether a given string belongs to a given language. The differences between levels of automata come from a specific implementation detail: The *memory system* required in order for them to be able to recognize their preferred patterns within strings.

To elaborate, each layer of automata can actually only recognize a certain subclass of languages. When a language is beyond the capacity of the automata layer, we then move on to a higher level automata. This ends with register machines which have the complexity potential of (ability to represent) arbitrary computable functions, but which also come with their own theoretical limitations—namely decidability.[6]

As stated in the philosophy section above, the novel idea of this essay is to present induction grammar that parallels function evaluation. I can now say specifically the intention here is to demonstrate grammar that corresponds to the regular and context-free levels of automata, and to show the potential for grammar at the higher levels along with the theoretical implications of these possibilities.

Construction

We've reached the other half of the evaluation correspondence: **construction**.

Since we're modelling function evaluation with automata, grammars, and languages, the correspondence with construction now comes from *function composition* which uses the insight that **chain composition** aligns quite well with the idea of a *string*:

$$f_0 f_1 f_2 \cdots f_{n-1} f_n \quad := \quad f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1 \circ f_0$$

Here we choose to perceive the individual functions as "characters" and use composition to sequence them into the string. Keep in mind we will need some way to check the **compositional coherence** of such strings, which is to say we will need to be able to check when such strings are even valid compositions in the first

[5]In traditional teachings of automata theory, instead of register machines the theoretical and alternative automata taught is that of the well known Turing machine: Such machines are conceptually simpler and easier to use when proving mathematical theorems about computability.

[6]Decidability is equivalent to the halting problem for which there is no single general solution. The idea is if we are going to compute a function we'd like to know if it will halt based on the input we give it. A language is recursively enumerable (semidecidable) if we know its recognizer function will halt with proper input. It may or may not halt given *incorrect* input, and we may even be able to prove as much case by case for each given function, but the halting problem says there's no unified proof for all such functions.

place. As for how these strings are read, we adhere to the cultural convention of reading from *left-to-right*, but it's certainly possible to read the other way as well:

$$\begin{aligned} {}_{\rightarrow}\, f_0 f_1 f_2 \cdots f_{n-1} f_n &= f_n \circ f_{n-1} \circ \ldots \circ f_2 \circ f_1 \circ f_0 \\ &= f_n f_{n-1} \cdots f_2 f_1 f_0 \,{}_{\leftarrow} \end{aligned}$$

In practice when the context is clear we will default to the left-to-right reading, but if it's *right-to-left* it will be made more explicit in the given discourse either in words or through this *arrow reading* notation.

In anycase, since our theory of construction is very much dependent upon the idea of composition, we will borrow concepts from category theory as it already has many well developed notions in this area. As I can't assume everyone is familiar with category theory, I will now give a quick summary of some basics.

Category Theory

Category theory is a foundational branch of mathematics which can act as an alternative to set theory. My explanation of it is informed by [5], although our notations differ a little. Category theory provides many valuable models for type theory and is increasingly becoming an active area of research in computing science because of this. We will focus on the computationally oriented aspects of the theory here.

Casually, a category $\mathcal{C}$ is made up of a class of objects and their morphisms, such that the morphisms have an operator $\circ$ which otherwise acts like our well behaved function composition operator. I will use this notation for both, but to signify the difference I will at times write function composition more specifically as:

$$\text{compose}(f, g) \;:=\; f \circ g$$

Otherwise, the morphism composition operator is well behaved with the following properties:

- Composition is associative: $(f \circ g) \circ h = f \circ (g \circ h)$.
- Composition has an identity: $f \circ \text{id} = f = \text{id} \circ f$.

Keep in mind I'm oversimplifying things here. To be technically accurate we have to consider the *domains* and *codomains* of morphisms to maintain their compositional coherence. Also, the above identity property makes it appear like there is only one identity morphism `id` per composition operator, but in fact there is one identity morphism $\texttt{id}_c$ per category object c, we only hide the index by convention.

Given such morphism composition we introduce two specializations which will be useful in the following:

$$\begin{aligned} \text{precompose}_f(g) &= f^{\circ}(g) &:=&\; g \circ f \\ \text{postcompose}_f(g) &= f_{\circ}(g) &:=&\; f \circ g \end{aligned}$$

These specializations are achieved by fixing either of the arguments (respectively) of our composition operator.

As with any other theory of math we will want to be able to compare our subjects of interest (here categories) using some kind of specialized *property preserving map.* In this context the equivalent idea is that of a *functor* which is traditionally signified by uppercase roman letters: $F : \mathcal{C} \to \mathcal{D}$. Beyond that, this theory diverges from most math[7] because it also finds a way not only to compare categories, but to compare functors. To do this, it uses the idea of a mapping between functors which it calls a *natural transformation*, where such transformations are themselves traditionally denotated by lowercase greek letters such as $\alpha : F \to G$.

Functors and natural transformations are very important in category theory in general, but are more specifically relevant to us because each behaves enough like the common intuitive idea we have of a function that we can talk about them having their own composition operators. In the case of functors, we reuse the function composition notation as the context tends to be clear enough that no ambiguity ensues:

$$F : \mathcal{C} \to \mathcal{D} \quad \text{and} \quad G : \mathcal{D} \to \mathcal{E} \quad \text{suggest} \quad (G \circ F) : \mathcal{C} \to \mathcal{E}$$

[7]The exception being homotopy and related theories which are in fact the origins for category theory.

In theory we could apply this same logic to natural transformations except that based on how they're defined they actually allow for two distinct equally valuable composition operators to which we give them their own notations:

$$\text{vertical composition:}\quad \alpha : F \to G \quad\text{and}\quad \beta : G \to H \quad\text{suggest}\quad (\beta \cdot \alpha) : F \to H$$

$$\text{horizontal composition:}\quad \alpha : F \to G \quad\text{and}\quad \beta : H \to K \quad\text{suggest}\quad (\beta \star \alpha) : (H \circ F) \to (K \circ G)$$

Vertical composition is analogous to our common idea of composition, but horizontal composition is pretty out there if you're not familiar with it. I won't go into the details as to how it actually works, just know it exists and is made possible because natural transformation domains and codomains (being functors) are themselves composable.

Okay, so we now kind of know some category theory, great! But how do these specific ideas of composition help us in particular?

The last major idea to introduce from category theory is that of a *monad*, which is an endofunctor $T : \mathcal{C} \to \mathcal{C}$ (a functor with equal domain and codomain) along with two of its own specialized natural transformations:

$$\begin{aligned} \text{unit:}\quad & \eta : \text{Id} \to T \\ \text{multiply:}\quad & \mu : T^2 \to T \end{aligned}$$

where Id: $\mathcal{C} \to \mathcal{C}$ is the identity endofunctor and T^2 is just shortform for $T \circ T$.

The relevance of a monad is it allows us to define a secondary composition operator for the morphisms of a given category. This is done by way of defining what's called a **Kleisli** category which is built using some given monad (T, η, μ). From this we then specify the secondary composition operator for morphisms as:

$$\text{endopose:}\quad f : A \to T(B) \quad\text{and}\quad g : B \to T(C) \quad\text{suggest}\quad (g \star f) : A \to T(C)$$

we name this operator **endopose**, the word being derived from the *endofunctor* with which it is is built.[8]

The big idea here for us is that wherever we encounter composition (or some baseline interpretation of it), and we also observe objects which don't quite compose but nearly do $A \to T(B),\ B \to T(C)$, we can often generate a secondary composition operator within the same context. I won't get into the math here, but this secondary monadic composition is just as well behaved as the original in terms of its own compositional coherence properties.[9]

To reiterate, we now have two composition operators within the same context:

$$\begin{array}{lllllll} \text{compose:} & f : A \to B & \text{and} & g : B \to C & \text{suggest} & (g \circ f) : A \to C \\ \text{endopose:} & f : A \to T(B) & \text{and} & g : B \to T(C) & \text{suggest} & (g \star f) : A \to T(C) \end{array}$$

though in practice we can have as many *derivative* composition operators as there are relevant monads.

Secondary composition parallels its primary, so it becomes quite reasonable to include its own specializations:

$$\begin{aligned} \text{preendopose}_f(g) &= f^\star(g) &:=& \quad g \star f \\ \text{postendopose}_f(g) &= f_\star(g) &:=& \quad f \star g \end{aligned}$$

Next, there is one other operator which is so deeply connected to composition it's also worth introducing here:

$$\begin{aligned} \text{apply}(f, x) \quad &: \quad (A \to B) \times A \ \to \ B \\ &:= \quad f(x) \\ &= \quad f \ \hookleftarrow\ x \end{aligned}$$

[8]This is not to downplay the importance of the underying monad (as there exist endofunctors which are not monads), rather it is more about designating a proper name for such secondary compositions. In particular the word *endopose* semantically aligns with the intended associations given that *endo-* means "within" and *poser* means "to place".

[9]These can be proven using the horozontal composition properties of the monad's natural transformations η, μ. For this reason, secondary composition inherits the horizontal composition notation $\star$ as this further alludes to its origins.

This **apply** operator takes a function f, and some value x and *applies* them to get their output value. The main connection for us is that composition can be defined out of application:

$$\begin{aligned} \text{compose}(g, f) \quad &: \quad (B \to C) \times (A \to B) \;\to\; (A \to C) \\ &:= \quad \text{apply}(g,\ \text{apply}(f, -)\,) \end{aligned}$$

where $(-)$ is the variable placeholder for the composite function's input.

Actually, application can also be defined out of composition so we can consider them alternatives, but for our purposes we will frequently use `apply` over `compose` because the `apply` operator ends up being more flexible for the grammars we want to build.[10] With that said, we will also often use them interchangeably when no confusion would otherwise arise.

Finally, given the duality between primary and secondary compositions (when they exist), there is a parallel to the `apply` operator called **bind**:[11]

$$\begin{aligned} \text{bind}(f, x) \quad &: \quad (A \to T(B)\,) \times T(A) \to T(B) \\ &:= \quad \mu(T(f)(x)) \\ &= \quad f\langle x \rangle \\ &= \quad f \;\dashv\; x \end{aligned}$$

In this case we can then properly define `endopose` as:

$$\begin{aligned} \text{endopose}(g, f) \quad &: \quad (B \to T(C)\,) \times (A \to T(B)\,) \;\to\; (A \to T(C)\,) \\ &:= \quad \text{bind}(g,\ \text{bind}(f,\ \eta(-)\,)\,) \end{aligned}$$

Lazy Construction

There's one last set of operators worth introducing in this methodology section which are key points of grammar when implementing what's known as the **lazy programming paradigm**—which itself is something we'll need in the following.

In functional programming languages such as LISP, *lazy evaluation* (also known as *normal order evaluation*) is the idea that function application is achieved by only evaluating the parts of the function that are actually used at the time that the function is called. In practice there are two operators needed to implement this paradigm:

$$\begin{aligned} \text{delay}(f, x) \quad &:= \quad (f, x) \\ \text{force}((f, x)) \quad &:= \quad f(x) \end{aligned}$$

their terminology comes from [4]. Lazy evaluation is achieved by using **delay** and **force** to decompose `apply`:

$$\text{apply} \;=\; \text{force} \circ \text{delay}$$

The operators `delay` and `force` are key, but for our purposes we will include a third:

$$\text{transit}(g) \quad := \quad g \circ \text{force} \quad = \; \text{force}^{\circ}(g)$$

I have named this function **transit** as for me it resembles logical *transitivity* in its use. Its value for us here is that it allows us to extend `force` so that our lazy evaluation can be applied to a wider range of grammar.

[10] By using the `apply` operator we can not only evaluate a function immediately by providing a given argument x, we can alternatively extend the function through composition allowing for the future possibility of further compositions. In the context of function construction we consider this a higher entropy design.

[11] The Haskell programming language has no name I'm aware of for `endopose`, but denotes it as $<\!=\!<$. It also makes strong use of the `bind` operator denoting it as $=\!\ll$. Such monadic theory is valuable in this line of programming languages as it allows their compilers to automate the definitions of many type-safe and user-friendly grammatical constructs such as **do** based off these and other monadic constructs such as `join`, `return`, and `fmap`.

Conditional Composition

We are finally past the methodology section, but before we can get to proper forms of automata induction we need to build up the endoposition operators for which they are defined. In particular these operators can be viewed as forms of **conditional composition**, and so this now becomes our focus.

The Operator Lattice

There are several operators that will be introduced in this section which can be summarized with the following lattice:

$$\begin{tikzcd}
 & & \text{if}_{(1,1)} \arrow[dl] \arrow[dr] & & \\
 & \text{hold}_{(1,2)} \arrow[dl] \arrow[dr] & & \text{cohold}_{(2,1)} \arrow[dl] \arrow[dr] & \\
\text{pend}_{(1,3)} \arrow[dr] & & \text{dihold}_{(2,2)} \arrow[dl] \arrow[dr] & & \text{copend}_{(3,1)} \arrow[dl] \\
 & \text{stem}_{(2,3)} \arrow[dr] & & \text{costem}_{(3,2)} \arrow[dl] & \\
 & & \text{distem}_{(3,3)} & &
\end{tikzcd}$$

This lattice represents a dependency narrative between these operators, for which we start with the well known **if** operator:

$$\text{if}\,(\textit{boolean},\ \textit{antecedent},\ \textit{consequent})$$

This construct returns *antecedent* when the *boolean* value is *true*, otherwise it returns *consequent* when it's *false*. This is the "conditional" in these composition operators. In terms of the lattice, the operator `if` is subscripted with $(1,1)$ which signifies it accepts 1 argument on its **antecedent side** and 1 on its **consequent side**. In particular the arguments it accepts are the *antecendent* and *consequent* themselves. The meaning of this will become clearer as we continue.

The next step up is to equip this conditional operator with what can be called aspects of composition. The following four functions **hold**, **cohold**, **pend**, **copend** are the first extensions of `if`:

```
(define (hold true? d e f)
        (if (true?) d
            (delay e f)))
```

```
(define (pend true? d e f g)
        (if (true?) d
            (delay
                (transit e)
                (delay f g))
        ))
```

```
(define (cohold true? d e f)
        (if (true?)
            (delay d e)
             f
        ))
```

```
(define (copend true? d e f g)
        (if (true?)
            (delay
                (transit d)
                (delay e f))
            g
        ))
```

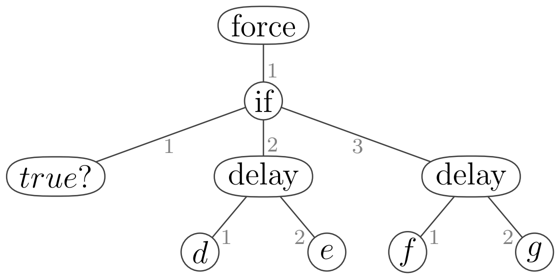
Truth be told these operators serve more as stepping stones and placeholders as I have not yet found other uses for them. I introduce them here because they create a smoother transition in building the remaining operators, and later when we perform certain optimizations they act as convenient boundary references.

As for their nomenclature, the word "hold" is intended to suggest connotations along the lines of "the *holding* of a transaction" (until certain conditions are met). The word "pend" is similar, but instead of a single transaction its word associations suggest "more than one operation is *pending*." As for the prefix "co-" it's actually analogous to traditional word use of the *sine* and *co-sine* trigonometric functions, or *product* and *co-product* in category theory: Strictly one could be defined by the other, but we have both because it's practical to have both.

Dihold

The first of our serious conditional composition operators is called **dihold**, and is defined as follows:

$$\text{dihold}(true?, d, e, f, g):$$



To contrast this operator with `hold` and `cohold` it instead "holds" on both sides of its conditional. This provides us with a unique opportunity to extend the *lazy paradigm* since either output will be a delay, which for us means there's no issue in forcing their respective compositions or evaluations. Also, notice in the above lattice that `dihold` is scripted with $(2, 2)$ and that here it takes 2 arguments on its *antecedent side* and 2 on its *consequent side*. This is the meaning behind this pattern, which will be used later when we optimize.

If we were to code `dihold` in the LISP programming style while also building on its predecessors it would be along these lines:

```
(define (dihold true? d e f g)
        (force
           (hold true? (delay d e) f g)
        ))
```

The Stem Operator

The next conditional composition operator is an extension of `dihold` designed to be more suitable for *continuation passing*, which is a valuable paradigm to us as it happens to be a known monad.

Intuitively, continuation passing works when you have a function $f(x, c)$ which not only takes its expected input x, it takes a *continuation* function for which the output is passed. Think of it like this: We take the function f and input value x and partially apply them. The internal definition of f is such that we get some partial output y, but because our function f is of continuation passing style we then pass this partial output to the continuation function:

$$f(x, c) \quad := \quad c(\hat{f}(x))$$

where $\hat{f}(x)$ represents the partial application output.

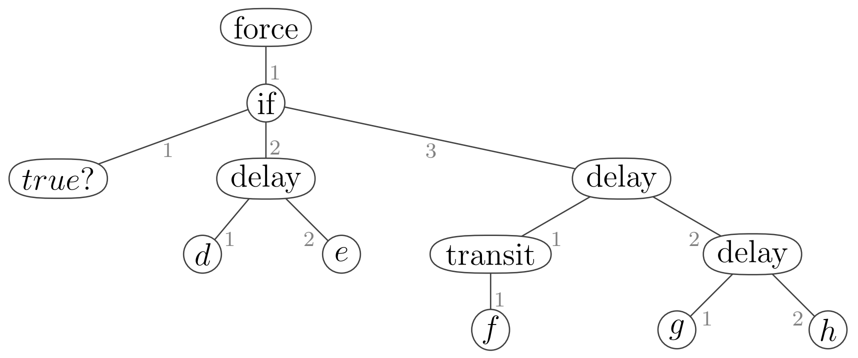
Next, as continuation passing style is known to be a monad, I present here an informal definition of what would be the *endoposition* of two continuation passing functions:

$$f(x, c_1(y)) \,\star\, g(y, c_2(z)) \quad := \quad f(\,x,\ \lambda y.g(y)(c_2(z))\,)$$

Here the use of the lambda symbol (λ) is as in the Lambda Calculus—a way to represent *anonymous* functions. As well, the $g(y)$ component is the partial application known as *currying*.

As for the definition of the `stem` operator, it is as follows:

stem($true?, d, e, f, g, h$):



In this case we have the same *branching* pattern as with `dihold`, but now if the boolean value is *false* and we return the *consequent* instead of the *antecedent* of the conditional `if`, we not only force the function and value pair g, h, we also pass them as input to the function f.

In LISP style, and based off our above `dihold` definition, `stem` would be coded as:

(define (stem $true?$ d e f g h)
    (dihold $true?$ d e (transit f) (delay g h))
)

This operator is interesting in many ways in its own right, in fact I call it the `stem` operator in analogy to *stem cells* in biology: As a function model it can specialize to several important compositional patterns—such as recursion—known in existing computational theories. Yet, for our purposes here its greatest value is in its ability to chain compose with itself.

To simplify notation a little, I now introduce `stem`'s conditional endopose operators.

Stem's CPose Operators

We start by taking the `stem` operator:

$$\text{stem}(\,policy?,\ break,\ arg_1,\ cont,\ next,\ arg_2\,)$$

Now, we refactor

$$arg_1,\ arg_2$$

and partially apply

$$policy?,\ break,\ next$$

to derive a function in continuation passing style:

$$\langle\,policy?,\ break,\ next\,\rangle(\,arg,\ cont\,) \quad := \quad \text{stem}(\,policy?,\ break,\ arg,\ cont,\ next,\ arg\,)$$

Note that $\langle\,policy?,\ break,\ next\,\rangle$ represents the function name. This function is in what I would call *conditional continuation passing style*. In particular, if we adhere to the rule that we don't apply or simplify the `stem` operator directly, then all the monadic rules of continuation passing hold. I mentioned *conditional endoposition* above, which I can now explain as being in the sense that once we allow ourselves to evaluate the internal `stem` operators, we then have to accept that any chain of conditionally endoposed functions may *break* before all of such functions are called.

From such monadic extensions we can now introduce `stem`'s **cpose** operators:

$$\begin{array}{llcl}
\langle \mathit{true?},\ \mathit{break},\ \mathit{next} \rangle & \underset{\texttt{call call}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ \mathit{break},\ -_{arg},\ \mathit{cont},\ \mathit{next},\ -_{arg}) \\
\langle \mathit{true?},\ \mathit{break},\ \mathit{next} \rangle & \underset{\texttt{call call}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ \mathit{break},\ -_{arg},\ \mathit{cont},\ \mathit{next},\ -_{arg}) \\
\langle \mathit{true?},\ \mathit{break},\ x \rangle & \underset{\texttt{call pass}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ \mathit{break},\ -_{arg},\ \mathit{cont},\ -_{arg},\ x) \\
\langle \mathit{true?},\ \mathit{break},\ \mathit{next},\ x \rangle & \underset{\texttt{call pose}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ \mathit{break},\ -_{w},\ \mathit{cont},\ \mathit{next},\ x) \\
\langle \mathit{true?},\ w,\ \mathit{next} \rangle & \underset{\texttt{pass call}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ -_{arg},\ w,\ \mathit{cont},\ \mathit{next},\ -_{arg}) \\
\langle \mathit{true?},\ w,\ x \rangle & \underset{\texttt{pass pass}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ -_{arg},\ w,\ \mathit{cont},\ -_{arg},\ x) \\
\langle \mathit{true?},\ w,\ \mathit{next},\ x \rangle & \underset{\texttt{pass pose}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ -_{break},\ w,\ \mathit{cont},\ \mathit{next},\ x) \\
\langle \mathit{true?},\ \mathit{break},\ w,\ \mathit{next} \rangle & \underset{\texttt{pose call}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ \mathit{next},\ -_{x}) \\
\langle \mathit{true?},\ \mathit{break},\ w,\ x \rangle & \underset{\texttt{pose pass}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ -_{next},\ x) \\
\langle \mathit{true?},\ \mathit{break},\ w,\ \mathit{next},\ x \rangle & \underset{\texttt{pose pose}}{\star\{\text{closing}\}}\ \mathit{cont} & := & \text{stem}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ \mathit{next},\ x)
\end{array}$$

This is a lot to take in, so let's go over the details.

For starters, given the informational complexity presented in these endoposition definitions, I have chosen to display them here while hiding the details of the right side argument (which is its own continuation passing function) replacing them with the condensed format of *cont*. Although I won't prove it here, these conditional endopose operators $\star$ really do satisfy the continuation passing composition properties described previously.[12]

Next, the generic or umbrella term for all the `cpose` operators in the above list is titled **closing**. The nomenclature here comes from the following naming convention regarding set *intervals* in math:

(a, b)	names an **open** interval.
$[\,a, b)$	names a **closing** interval.
$[\,a, b\,]$	names a **closed** interval
$(a, b\,]$	names an **opening** interval

The idea is, starting with an *open* interval, if we were to begin *closing* it we would do so from left-to-right which coincides with its conventional read order. Similarly when we start with a *closed* interval and we begin *opening* it we do so from left-to-right as well. Ideas of "openness" and "closedness" are relevant here in the sense that these `cpose closing` operators are *closed to continuing* when the boolean value is *true*, and are otherwise *open to continuing* when it's *false*.

Beyond this, each `cpose` operator is bottom-scripted with two words corresponding to the branching behaviours of the endoposition: They indicate how we compose the operator's input. For example **call call** indicates that when we break (based on `stem`'s definition) we do so by *calling* the *break* function to the as of yet named value $-_{arg}$. Of the two bottom-script words, this behaviour is specified by the left `call`. If instead this `stem` operator continues, it does so by *calling* the function *next* to our unnamed input $-_{arg}$. This is specified by the bottom-script right `call`.

To contrast `call call`, the **pass call** operator acts similarly, but upon breaking it instead *passes* the *break* function to the $-_{arg}$ value, which in this context must be a function of its own. In fact this brings up another point, that the standard interpretation when evaluating these `cpose`s is that of composing functions, but in practice we often also use it to express the application of function and value. The terminology for *call* and *pass* come from the composition specializations previously introduced:

$$\begin{array}{rcccll}
\text{precompose}_f(g) & = & f^{\circ}(g) & := & g \circ f & (\text{“}f \text{ is } \textit{passed} \text{ to } g\text{”}) \\
\text{postcompose}_f(g) & = & f_{\circ}(g) & := & f \circ g & (\text{“}f \text{ is } \textit{called} \text{ for } g\text{”})
\end{array}$$

[12] I have worked out a pencil and paper proof myself, it's fairly tedious, but true.

Finally, if you'll notice, some of the above `cpose`s contain the **pose** modifier instead of `call` or `pass`. In practice the situation occasionally arises where we don't actually want to call or pass the $-_{arg}$ input value, but simply want to return a fixed composition or application. In such cases we make use of `pose`. Its name being short for the baseline com*pose* operator.

This whole naming system might seem a bit elaborate at the moment, but it really does make things simpler in the long run: Not only will it come up again with the remaining conditional endoposes, but it will also be used to standardize the nomenclature for a class of automata induction notation introduced later on. In anycase, the thing to note is that the above cpose operators are really just variations of each other, all of which afford us the ability to chain compose the `stem` operator with itself, and even more so, alternative instances of itself.

So, we've explored the naming system, but just how can these `cpose` operators lead us to better forms of *conditional composition*? At this point the easiest explanation might actually be just to show an example, so here goes:

$$\begin{array}{ll}
\langle\, true?_0,\ break_0,\ f_0\,\rangle & \underset{\texttt{call call}}{\star\{\text{closing}\}} \\
\langle\, true?_1,\ break_1,\ f_1\,\rangle & \underset{\texttt{call call}}{\star\{\text{closing}\}} \\
\langle\, true?_2,\ break_2,\ f_2\,\rangle & \underset{\texttt{call call}}{\star\{\text{closing}\}} \\
\qquad\quad\vdots & \\
\langle\, true?_n,\ break_n,\ f_n\,\rangle & \underset{\texttt{call call}}{\star\{\text{closing}\}} \\
\ \text{id}\ \ x &
\end{array}$$

Here `id` doesn't add anything meaningful to the composition except that by convention we are using it to represent the halting of what is otherwise a continuation passing process. The x value following `id` is applied at the very end once the final *composed* function is built. Beyond this, the major thing to note about such a pattern is that in the special case when all its *true*? values fail, we end up with a chain composition:

$$f_n \circ f_{n-1} \circ \ldots \circ f_1 \circ f_0\,(x)$$

To understand this let's translate the entire construct back to its definition:

$$\begin{array}{lcl}
 & & \text{stem}(\,true?_0,\ break_0,\ x,\ cont_0,\ f_0,\ x\,) \\
cont_0 & = & \text{stem}(\,true?_1,\ break_1,\ -_{arg},\ cont_1,\ f_1,\ -_{arg}\,) \\
cont_1 & = & \text{stem}(\,true?_2,\ break_2,\ -_{arg},\ cont_2,\ f_2,\ -_{arg}\,) \\
 & \vdots & \\
cont_{n-1} & = & \text{stem}(\,true?_n,\ break_n,\ -_{arg},\ cont_n,\ f_n,\ -_{arg}\,) \\
cont_n & = & \text{id}
\end{array}$$

On the one hand, we can start at the end with the identity function `id` and chain these `cposes` from that direction. We can do this on account that these cposes are (unconditionally) *right associative*. From this theoretical underpinning, we then properly define each

$$cont_n,\ cont_{n-1},\ \ldots,\ cont_1,\ cont_0$$

in sequence. Based off the definition of `stem` this should leave us with a single (if not large) function to which we can finally apply the value x. How then does this turn into a chain composition?

So on the other hand, as an alternative strategy we could have started out from the top of the sequence and tried composing that way. So long as we compose symbolically without evaluating the internal `stem` operators this works, but otherwise these `cpose` operators start to behave a lot stranger than we'd like. In general the forward approach is less revealing.

The exception here is when we restrict ourselves to these expressions where we know we will never break—where the *true*? values are always *false*: In such cases we would start by checking the boolean value of the first `stem` in the chain, and we'd discover the *true*? value to fail. As such, and given the nature of `stem`'s definition as a conditional, we wouldn't actually need to compose the whole operator with the next `stem` in line. With such foreknowledge we could instead dump the excess baggage (so to speak), and continue applying what remains. That's how we get the above chain composition.

In fact, given the regularity of the interpretation here we can simplify this whole sequence into the following formal grammatical pattern:

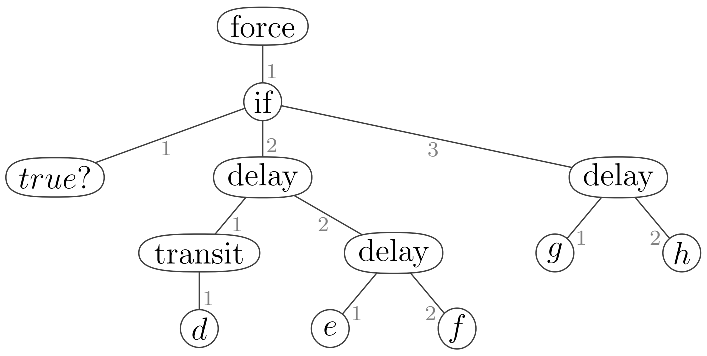
closing	**call**	**call**	x
$true?_0$	$break_0$	f_0	
$true?_1$	$break_1$	f_1	
$true?_2$	$break_2$	f_2	
$\vdots$	$\vdots$	$\vdots$	
$true?_n$	$break_n$	f_n	
closed			

where **closed** (at the bottom) is a special `cpose` that halts regardless of its *boolean* input value. As well, if its options aren't specified it defaults to the identity `id`.

The Costem Operator

As with `hold` and `pend`, `stem` also has a dual operator:

costem($true?, d, e, f, g, h$):



Like its predecessors **costem** is provided for convenience and performance (potentially; in some situations). In LISP it would be encoded as:

```
(define (costem true? d e f g h)
        (dihold true? (transit d) (delay e f) g h)
)
```

If one wishes we could alternatively implement `costem` using its dual by negating the *true*? value. Either way, and given this duality, `costem` has its own `cpose` operators in parallel to `stem`:

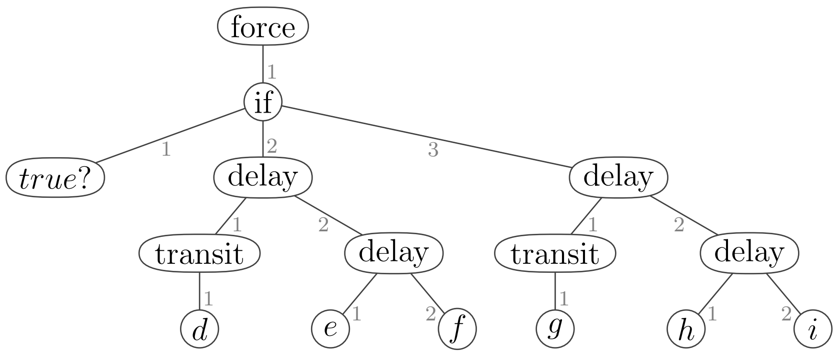
Costem's CPose Operators

$\langle\, true?,\ next,\ break\, \rangle$	$\underset{\text{call call}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ next,\ -_{arg},\ break,\ -_{arg}\,)$
$\langle\, true?,\ next,\ w\, \rangle$	$\underset{\text{call pass}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ next,\ -_{arg},\ -_{arg},\ w\,)$
$\langle\, true?,\ next,\ break,\ w\, \rangle$	$\underset{\text{call pose}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ next,\ -_{x},\ break,\ w\,)$
$\langle\, true?,\ x,\ break\, \rangle$	$\underset{\text{pass call}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ -_{arg},\ x,\ break,\ -_{arg}\,)$
$\langle\, true?,\ x,\ w\, \rangle$	$\underset{\text{pass pass}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ -_{arg},\ x,\ -_{arg},\ w\,)$
$\langle\, true?,\ x,\ break,\ w\, \rangle$	$\underset{\text{pass pose}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ -_{next},\ x,\ break,\ w\,)$
$\langle\, true?,\ next,\ x,\ break\, \rangle$	$\underset{\text{pose call}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ next,\ x,\ break,\ -_{w}\,)$
$\langle\, true?,\ next,\ x,\ w\, \rangle$	$\underset{\text{pose pass}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ next,\ x,\ -_{break},\ w\,)$
$\langle\, true?,\ next,\ x,\ break,\ w\, \rangle$	$\underset{\text{pose pose}}{\star\{\text{opening}\}}\ cont$	$:=$	$\text{costem}(\, true?,\ cont,\ next,\ x,\ break,\ w\,)$

The Distem Operator

Finally, we end this section on conditional composition with the **distem** operator:

$$\text{distem}(true?, d, e, f, g, h, i):$$



which in LISP becomes:

(define (distem *true*? *d e f g h i*)
(stem *true*? (transit *d*) (delay *e f*) *g h i*))
)

The importance of the `distem` operator is that it extends *continuation passing* to both sides of the conditional: Instead of the possibility of *break*ing as with `stem` or `costem` it continues no matter what. In the special case where both continuations are the same, this allows for *chain alternation* which greatly increases our expressivity to build functions in the long run. As for the situation where the two continuations are in fact different (known as *chain branching*), it's certainly possible, but we can also achieve the same effect with variables and a memory system for which to split up our grammatical source code into smaller parts. This is likely a better design.

In anycase, when restricted to a single continuation, `distem` like its predecessor `stems` also has its own `cpose` operators.

Distem's CPose Operators

$$
\begin{array}{llcl}
\langle\, \mathit{true?},\ \mathit{next}_1,\ \mathit{next}_2 \,\rangle & \underset{\texttt{call call}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ -_{arg},\ \mathit{cont},\ \mathit{next}_2,\ -_{arg}\,) \\
\langle\, \mathit{true?},\ \mathit{next}_1,\ x_2 \,\rangle & \underset{\texttt{call pass}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ -_{arg},\ \mathit{cont},\ -_{arg},\ x_2\,) \\
\langle\, \mathit{true?},\ \mathit{next}_1,\ \mathit{next}_2,\ x_2 \,\rangle & \underset{\texttt{call pose}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ -_{x_1},\ \mathit{cont},\ \mathit{next}_2,\ x_2\,) \\
\langle\, \mathit{true?},\ x_1,\ \mathit{next}_2 \,\rangle & \underset{\texttt{pass call}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ -_{arg}\,) \\
\langle\, \mathit{true?},\ x_1,\ x_2 \,\rangle & \underset{\texttt{pass pass}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ x_1,\ \mathit{cont},\ -_{arg},\ x_2\,) \\
\langle\, \mathit{true?},\ x_1,\ \mathit{next}_2,\ x_2 \,\rangle & \underset{\texttt{pass pose}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ -_{next_1},\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ x_2\,) \\
\langle\, \mathit{true?},\ \mathit{next}_1,\ x_1,\ \mathit{next}_2 \,\rangle & \underset{\texttt{pose call}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ -_{x_2}\,) \\
\langle\, \mathit{true?},\ \mathit{next}_1,\ x_1,\ x_2 \,\rangle & \underset{\texttt{pose pass}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ -_{next_2},\ x_2\,) \\
\langle\, \mathit{true?},\ \mathit{next}_1,\ x_1,\ \mathit{next}_2,\ x_2 \,\rangle & \underset{\texttt{pose pose}}{\star\{\text{open}\}}\ \mathit{cont} & := & \text{distem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ x_2\,)
\end{array}
$$

Induction Automata Correspondences

We now have enough of a basis to understand our first class of induction operators.

Regular Induction

The first layer of functions that we can create an inductive grammar for are what I would call **regular functions**. The local correspondence is with *regular automata*, which informs the naming convention here. As for regular automata themselves, we will in fact be making use of an equivalent known as *regular expressions*.

As a quick review, regular expressions are recursively defined objects. Starting with their initial specification they are first and foremost strings (made up of characters), but are otherwise built from such strings using three core operators:

1. **catenation** - If a, b are regular expressions, then ab is a regular expression.
2. **alternation** - If a, b are regular expressions, then $(a\,|\,b)$ is a regular expression.
3. **repetition** - If a is a regular expression, then a^* is a regular expression.

If you've used regular expressions before you might protest that there are many more expression operators provided within actual programming languages. The majority of these are either syntactic sugar (convenience operators) derived from the above three, or they are custom extensions which may be useful in practice but otherwise deviate from the underlying mathematical theory. Either way, we will not delve into such variations here.

Finite state automata at their most basic are *recognizers*, also known as *acceptors*. Without going into the underlying details of how they work, regular automata are machines with a finite and fixed memory system which they use to recognize strings given to them. In particular they are only able to recognize those strings which were constructed by the above operators (given some initial alphabet).

The point for us though is not to be able to read strings of chain composed functions, but to write them.

Catenation

The writer form of catenation is straightforward function composition, which is immediately achieved using the `apply` operator:

$$ {}_{\rightarrow}\, f_0 f_1 \cdots f_{n-1} f_n \quad := \quad \text{apply}(f_n,\ \text{apply}(f_{n-1},\ \text{apply}(\ldots,\ \text{apply}(f_1,\ \text{apply}(f_0,\ -))\ldots))) $$

We could of course also make use of `bind` operators when their respective monads exist for them:

$$_{\rightarrow}\, f_0 f_1 \cdots f_{n-1} f_n \quad := \quad \mathrm{bind}(f_n,\ \mathrm{bind}(f_{n-1},\ \mathrm{bind}(\ldots,\ \mathrm{bind}(f_1,\ \mathrm{bind}(f_0,\ \eta(-)))\ldots)))$$

In this sense catenation parallels tuple construction by means of recursively nested pairs:

$$\begin{aligned}(f_n,\, f_{n-1}, \ldots,\, f_1,\, f_0) \quad &:= \quad \mathrm{cons}(f_n,\ \mathrm{cons}(f_{n-1},\ \mathrm{cons}(\ldots,\ \mathrm{cons}(f_1,\ \mathrm{cons}(f_0,\, \mathrm{null}))\ldots))) \\ &= \quad (f_n,\ (f_{n-1},\ (\ldots,(f_1,\ (f_0,\ \mathrm{null}\,))\ldots)))\end{aligned}$$

which might be something worth noting.

So far we've only demonstrated the *horizontal* notation for chain composition, but we will also want a *vertical* notation in the long run:

$$\begin{array}{l}\textbf{chain call}\ \ x \\ function_0 \\ function_1 \\ function_2 \\ \quad\vdots \\ function_n \\ \textbf{done}\end{array}$$

Unlike horizontal chain composition there's no need for *read* directions—the only direction here being *top-down*.[13] There is however a variant which composes in reverse:

$$\begin{array}{l}\textbf{chain pass}\ \ \mathrm{id} \\ function_n \\ function_{n-1} \\ function_{n-2} \\ \quad\vdots \\ function_0 \\ \textbf{done}\end{array}$$

Either way, this style can be considered more user-friendly to read when the composite functions themselves take up a lot more horizontal spacing (which in practice happens often enough).

Alternation

The meaning of alternation is that of *alternatives*: It parallels the logical idea of the **or** operator ($\vee$), as well as the type theoretic *coproduct*. In the theory of automata it means that any regular expression possessing the term $(a\,|\,b)$ has an *alternation site* which will be recognized. For example if we're searching a text for the string "Fourier" and we want to include versions with both lower and uppercase lettering for the initial character we could specify:

$$(\,\mathrm{F}\,|\,\mathrm{f}\,)\mathrm{ourier}$$

which when translated into a regular automata would now recognize both possibilities:

$$\{\ \mathrm{Fourier}\ ,\ \mathrm{fourier}\ \}$$

In the theory being offered here, when interpreting such strings as function compositions this regular expression $(\,\mathrm{F}\,|\,\mathrm{f}\,)$ourier would then be read as two independent chain constructions. As such, using this style of grammar alone we can already specify two alternative functions within a chain composition.

In practice though, this style of expression is not directly as useful as one would hope, especially as the set of possible strings doubles in size with every additional alternation site. In fact it's more than likely

[13] An exception comes when we take this notation to its logical extreme and fill up an entire page. If we continue such a composition by making further columns on the same page, we will then need to specify a horizontal read direction once again.

in our applications we would want to resolve such a specification down to a single chain within the given alternatives.

Now in terms of retooling alternation for function construction, we rely on the previously introduced `distem` operator. If we go back to `distem`'s `cpose` operators and their definitions we are only really missing a single component, namely the boolean value *true*?. As such, we equip our alternation grammar with this variable:

$$\underset{true?}{(\,\mathrm{F}\,|\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$$

In this context though I feel it's more appropriate to relabel this boolean value as *policy*? :

$$\underset{policy?}{(\,\mathrm{F}\,|\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$$

which gives a stronger connotation of *choice making* when it comes to our construction.

This notation is incomplete though. It currently corresponds to `distem`'s `open cpose`, but we're lacking the appropriate modifiers. As example let's take `call call`, which we can then specify as:

$$\underset{\substack{\text{call call}\\ policy?}}{(\,\mathrm{F}\,|\,\mathrm{f}\,)}\;\mathrm{ourier}_{\leftarrow} \qquad \text{or} \qquad \overset{\text{call call}}{\underset{policy?}{(\,\mathrm{F}\,|\,\mathrm{f}\,)}}\mathrm{ourier}_{\leftarrow}$$

At this point I find the notation lacking in general elegance. In particular I favor use of the specialized composition operators:

$$\underset{policy?}{(\,\mathrm{F}_\circ\,|\,\mathrm{f}_\circ\,)}\mathrm{ourier}_{\leftarrow}$$

which works if you recall such specializations as being:

$$\begin{array}{rcccll}
\mathrm{precompose}_f(g) & = & f^\circ(g) & := & g \circ f & \qquad (\text{“}f \text{ is } \textit{passed} \text{ to } g\text{”}) \\
\mathrm{postcompose}_f(g) & = & f_\circ(g) & := & f \circ g & \qquad (\text{“}f \text{ is } \textit{called} \text{ for } g\text{”})
\end{array}$$

In this case the alternatives are as follows:

open call call: $\underset{policy?}{(\,\mathrm{F}_\circ\,|\,\mathrm{f}_\circ\,)}\mathrm{ourier}_{\leftarrow}$

open call pass: $\underset{policy?}{(\,\mathrm{F}_\circ\,|\,\mathrm{f}^\circ\,)}\mathrm{ourier}_{\leftarrow}$

open call pose: $\underset{policy?}{(\,\mathrm{F}_\circ\,|\,\mathrm{f}\;x)}\mathrm{ourier}_{\leftarrow}$

open pass call: $\underset{policy?}{(\,\mathrm{F}^\circ\,|\,\mathrm{f}_\circ\,)}\mathrm{ourier}_{\leftarrow}$

open pass pass: $\underset{policy?}{(\,\mathrm{F}^\circ\,|\,\mathrm{f}^\circ\,)}\mathrm{ourier}_{\leftarrow}$

open pass pose: $\underset{policy?}{(\,\mathrm{F}^\circ\,|\,\mathrm{f}\;x)}\mathrm{ourier}_{\leftarrow}$

open pose call: $\underset{policy?}{(\,\mathrm{F}\;x\,|\,\mathrm{f}_\circ\,)}\mathrm{ourier}_{\leftarrow}$

open pose pass: $\underset{policy?}{(\,\mathrm{F}\;x\,|\,\mathrm{f}^\circ\,)}\mathrm{ourier}_{\leftarrow}$

open pose pose: $\underset{policy?}{(\,\mathrm{F}\;x_1\,|\,\mathrm{f}\;x_2)}\mathrm{ourier}_{\leftarrow}$

On the other hand, if we want to reduce the number of *potential* parentheses when using this notation within our applications we can attach the composition circles $\circ$ to the alternation bar symbols $|$ directly:

open call call: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}_{\circ}|_{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

open call pass: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}_{\circ}|^{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

open call pose: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}_{\circ}|\,\mathrm{f}\,x)}\mathrm{ourier}_{\leftarrow}$

open pass call: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}^{\circ}|_{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

open pass pass: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}^{\circ}|^{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

open pass pose: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}^{\circ}|\,\mathrm{f}\,x)}\mathrm{ourier}_{\leftarrow}$

open pose call: $\underset{\textit{policy?}}{(\,\mathrm{F}\,x\,|_{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

open pose pass: $\underset{\textit{policy?}}{(\,\mathrm{F}\,x\,|^{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

open pose pose: $\underset{\textit{policy?}}{(\,\mathrm{F}\,x_1\,|\,\mathrm{f}\,x_2)}\mathrm{ourier}_{\leftarrow}$

This actually ties back to the nomenclature introduced with the `stem` and `costem cposes`, and although technically it moves us slightly outside of standard automata theory, `stem`'s correspondences are as follows:

closing call call: $\underset{\textit{policy?}}{[\,\mathrm{F}\,{}_{\circ}|_{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

closing call pass: $\underset{\textit{policy?}}{[\,\mathrm{F}\,{}_{\circ}|^{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

closing call pose: $\underset{\textit{policy?}}{[\,\mathrm{F}\,{}_{\circ}|\,\mathrm{f}\,x)}\mathrm{ourier}_{\leftarrow}$

closing pass call: $\underset{\textit{policy?}}{[\,\mathrm{F}\,{}^{\circ}|_{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

closing pass pass: $\underset{\textit{policy?}}{[\,\mathrm{F}\,{}^{\circ}|^{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

closing pass pose: $\underset{\textit{policy?}}{[\,\mathrm{F}\,{}^{\circ}|\,\mathrm{f}\,x)}\mathrm{ourier}_{\leftarrow}$

closing pose call: $\underset{\textit{policy?}}{[\,\mathrm{F}\,x\,|_{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

closing pose pass: $\underset{\textit{policy?}}{[\,\mathrm{F}\,x\,|^{\circ}\,\mathrm{f}\,)}\mathrm{ourier}_{\leftarrow}$

closing pose pose: $\underset{\textit{policy?}}{[\,\mathrm{F}\,x_1\,|\,\mathrm{f}\,x_2)}\mathrm{ourier}_{\leftarrow}$

with `costem`'s correspondences being symmetric:

opening call call: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}_{\circ}|_{\circ}\,\mathrm{f}\,]}\mathrm{ourier}_{\leftarrow}$

opening call pass: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}_{\circ}|^{\circ}\,\mathrm{f}\,]}\mathrm{ourier}_{\leftarrow}$

opening call pose: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}_{\circ}|\,\mathrm{f}\,x]}\mathrm{ourier}_{\leftarrow}$

opening pass call: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}^{\circ}|_{\circ}\,\mathrm{f}\,]}\mathrm{ourier}_{\leftarrow}$

opening pass pass: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}^{\circ}|^{\circ}\,\mathrm{f}\,]}\mathrm{ourier}_{\leftarrow}$

opening pass pose: $\underset{\textit{policy?}}{(\,\mathrm{F}\,{}^{\circ}|\,\mathrm{f}\,x]}\mathrm{ourier}_{\leftarrow}$

opening pose call: $\underset{\textit{policy?}}{(\,\mathrm{F}\,x\,|_{\circ}\,\mathrm{f}\,]}\mathrm{ourier}_{\leftarrow}$

opening pose pass: $\underset{\textit{policy?}}{(\,\mathrm{F}\,x\,|^{\circ}\,\mathrm{f}\,]}\mathrm{ourier}_{\leftarrow}$

opening pose pose: $\underset{\textit{policy?}}{(\,\mathrm{F}\,x_1\,|\,\mathrm{f}\,x_2]}\mathrm{ourier}_{\leftarrow}$

These are the horizontal notations for our alternation operators. There are also vertical notations analogous to the ones for the above catenation operators. In particular we reuse the grammatical forms introduced in the conditional composition section, for example the following horizontal expression:

$$x\,_{\rightarrow}\,\underset{\textit{policy?}_0}{[\,\textit{break}_0\ {}_{\circ}|_{\circ}\,f_0)}\ \underset{\textit{policy?}_1}{[\,\textit{break}_1\ {}_{\circ}|_{\circ}\,f_1)}\ \underset{\textit{policy?}_2}{[\,\textit{break}_2\ {}_{\circ}|_{\circ}\,f_2)}\ \ldots\ \underset{\textit{policy?}_n}{[\,\textit{break}_n\,{}_{\circ}|_{\circ}\,f_n)}$$

translates into the following vertical form:

$$\begin{array}{llll}
\textbf{closing} & \textbf{call} & \textbf{call} & x \\
\mathit{policy?}_0 & \mathit{break}_0 & f_0 & \\
\mathit{policy?}_1 & \mathit{break}_1 & f_1 & \\
\mathit{policy?}_2 & \mathit{break}_2 & f_2 & \\
\vdots & \vdots & \vdots & \\
\mathit{policy?}_n & \mathit{break}_n & f_n & \\
\textbf{closed} & & &
\end{array}$$

Repetition

In regular expression theory the repetition operator is known as the *Kleene star* operator, which as a recognizer allows us to match against repetitive strings:

$$a^* \quad := \quad \{\,\epsilon,\ a,\ aa,\ aaa,\ aaaa,\ \ldots\}$$

where ϵ is the empty string. Use of this *repeat* operator can be surprisingly effective, for example we need only it and alternation to generate all finite length binary strings:

$$(\,0\,|\,1\,)^* \quad := \quad \{\,\epsilon,\ 0,\ 1,\ 00,\ 01,\ 10,\ 11,\ 000,\ 001,\ 010,\ \ldots\}$$

As powerful an operation as this is, how can we convert it to be applicable to function construction? For example how do we interpret the repetition of a single function f ?

This operator works by means of composition just like the other regular expressions:

$$\begin{array}{rcl}
f^* & \sim & \mathrm{id} \\
 & \text{or} & f \\
 & \text{or} & f \circ f \\
 & \text{or} & f \circ f \circ f \\
 & \text{or} & f \circ f \circ f \circ f \\
 & \vdots & \quad\vdots
\end{array}$$

but in this case the empty string ϵ is interpeted as the identity function `id`. Otherwise, as with alternation this notation has limited expressivity until we can narrow things down to a specific instance (rather than the whole of the language it represents). To do this, we distinguish individual strings by their lengths ℓ:

$$(F)^{\ell}\text{ourier}_{\leftarrow}$$

which upon first glance might appear anti-climactic, but once we combine repetition with alternation we can actually achieve a reasonable level of expressivity, for example:

$$(f \underset{p}{\circ|\circ} g)^5$$

which would expand to

$$(f \underset{p_0}{\circ|\circ} g)(f \underset{p_1}{\circ|\circ} g)(f \underset{p_2}{\circ|\circ} g)(f \underset{p_3}{\circ|\circ} g)(f \underset{p_4}{\circ|\circ} g)$$

In fact we can even take things further, but for the more interesting versions of repetition we will need to solve the signature problem (which we'll get to soon enough).

The vertical notation for this repetition operator is an extension of the previous operator's vertical notations. In this case we *prepend* the keyword **repeat** followed by a number m which tells us how many times we want to repetitively compose the initial operator, for example:

$$\begin{array}{lll}
\textbf{repeat}\ \ m\ \ x & & \\
\textbf{closing} & \textbf{call} & \textbf{call} \\
\mathit{policy?}_0 & \mathit{break}_0 & f_0 \\
\mathit{policy?}_1 & \mathit{break}_1 & f_1 \\
\mathit{policy?}_2 & \mathit{break}_2 & f_2 \\
\vdots & \vdots & \vdots \\
\mathit{policy?}_n & \mathit{break}_n & f_n \\
\textbf{closed} & &
\end{array}$$

The proper interpretation here is to use this induction grammar to first construct the function for which we're passing the value x. More specifically, this means we start with $n + 1$ distinct *policy*? values:

$$\mathit{policy?}_0,\ \mathit{policy?}_1,\ \mathit{policy?}_2,\ \ldots\ \mathit{policy?}_n$$

which expand to $m(n + 1)$ such *policy*? values, and then we evaluate. This doesn't make a difference when such values are constant, but becomes especially meaningful when they are replaced with functions of their own. Also, in addition to a finite valued m, we can even specify it to be infinity ∞, but in this case we would expand it under an *only-as-needed* lazy style, for which we would also require guarantees in advance that the algorithm would eventually halt.

The Regular Pumping Lemma

So far we've been discussing the practical considerations of finite automata and their corresponding grammars. We haven't yet discussed their theoretical limitations. To that end, by going back to the original automata theory we are provided with **the pumping lemma for regular languages**, which states:

> Let $\mathcal{L}$ be a regular language with strings $w, x, y, z \in \mathcal{L}$ such that $wxy = z$ and $x \neq \epsilon$. There exists a natural number $n \in \mathbb{N}$ (dependent only on $\mathcal{L}$) such that if length$(z) \geq n$, and length$(wx) \leq n$, then for all $k \in \mathbb{N}$ we have:
>
> $$wx^k y \in \mathcal{L}$$

Without going heavily into the details of regular languages, the pumping lemma is a consequence of the finite memory system of their corresponding automata. In particular, if such a finite state machine has n states (size n memory) and the string we're testing against has length greater than n, then by the pigeonhole counting principle two or more characters in the string must be recognized by the same state.

The big realization here is that in the case such an automata needs to be able to distinguish the locations of two characters as part of its recognition algorithm, such information actually gets lost through these pigeonholes: It's like adding $3 + 4$, if our memory system only kept the end result 7 it wouldn't be able to recover the addends after the fact if they were later needed, for example we could guess $2 + 5$ instead with no way of knowing. Hence, there are many patterns of strings that regular automata simply don't have the capacity to recognize.

This lemma not only gives us a criterion to test if a particular chain composition of functions is a regular function, it also demonstrates the need to move beyond **regular induction**. With that said, we're still not quite ready to introduce the context-free variety: Before we do, we will need a few additional prerequisites which unfortunately could not have been introduced in the previous sections. As such, it's now time to revisit and finally solve the signature problem.

The Signature Problem

I had previously stated that the signature problem arises from the desire to *create variations within the signatures of functions.* More specifically it arises from wanting to create variations within **subsignatures** of functions. For example we had previously defined the `stem` operator which I now re-present in its LISP form:

```
(define (stem policy? d e f g h)
        (dihold policy? d e (transit f) (delay g h))
)
```

Notice here the *policy*? variable is a boolean value which is either *true* or *false* ? What if instead we wanted to turn it into a boolean function accepting the input h from the main signature:

```
(define (stem policy? d e f g h)
        (dihold (policy? h) d e (transit f) (delay g h))
)
```

Can you spot the difference in these two definitions?

We now have a small variation which semantically creates an entirely different operator than our original `stem`. In such cases do we then have to recode from scratch every time we make a small change to its sub-signatures? For example we might alternatively have a *policy*? function that accepts the following signature as its input:

$$(\textit{policy}?\ e\ g)$$

All in all there are $2^5 = 32$ possible variations based on whether a given variable belongs to our desired signature or not. Frankly that's a lot of *nearly identical* functions to have to hand code!

As for solving this problem there are a few common approaches: First, we could encapsulate the main function (or operator) within another superficial function and set the variable values accordingly:

```
(define (superficial policy? d e f g h)
        (stem (policy? h) d e f g h)
)
```

but this still doesn't help us when it comes to the other 31 variations. The second option is we could encapsulate *policy*? within its own superficial function when implementing the main function, and then disregard argument variables as needed:

```
(define (stem policy? d e f g h)
        (dihold (superficial′ policy? d e f g h) d e (transit f) (delay g h))
)
```

where for example:

```
(define (superficial′ policy? d e f g h)
        (policy? h)
)
```

but this runs into the same problem in terms of the remaining variations.[14]

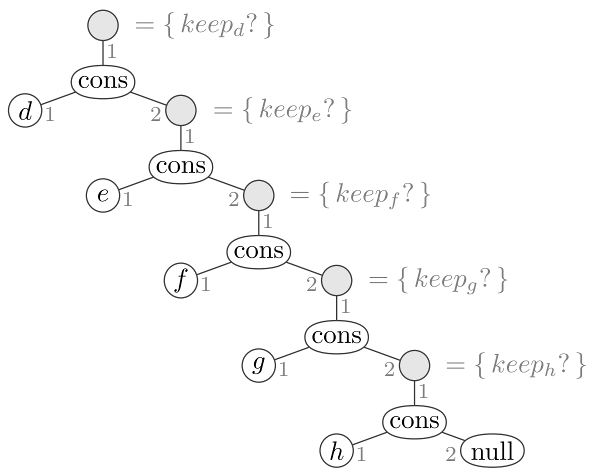
As such, how do we go about solving our signature problem? Sticking with the example signature:

$$(d,\ e,\ f,\ g,\ h)$$

which we now view as a tuple, we want in this case a grammatical operator that accepts policies determining the variables to *keep* and then builds the tuple accordingly. With that said, I would like to take this opportunity to practice the primary and secondary modelling approaches to operator design discussed in the methodology section. In that spirit, our first attempt at implementing this grammatical operator is to take the secondary modelling approach:

[14] Neither approach is the worst compromise, but if we survey how memory systems tend to be implemented we find extra work would be done in storing the unused variable value *bindings*. It's possible that in a given practical context we might have a nice compiler that will optimize such details out, but a user also shouldn't be forced to rely on that assumption—for example there are *mission critical* contexts where every computation counts and which might necessitate more refined control.

signature ($keep_d?,\ keep_e?,\ keep_f?,\ keep_g?,\ keep_h?$):



Here each *keep*? variable is meant to be set to either `id` or `cdr`, where `id` corresponds to *keeping* the variable and `cdr` corresponds to *dropping* it.

This approach is considered a secondary model because there are plenty of functions we could have substituted other than `id` or `cdr` having nothing to do with building tuples, but they'd be valid nonetheless. In this case such a pattern represents an entire class of functions. Regardless, you'll notice the inefficiencies in this style of modelling as there are a lot of unnecessary calculations within any actual tuple building function we create.

As I had mentioned in the methodology section this approach is a good place to start as it still tells us the bare minimum of components that a more refined primary model would likely require. To that end let's now turn to such a primary:

induct ⟨*induct_name*⟩ *func_name* $keep_d?$ $keep_e?$ $keep_f?$ $keep_g?$ $keep_h?$:

define *func_name*

id

open	**pass**	**id**
$keep_d?$	cons d	
$keep_e?$	cons e	
$keep_f?$	cons f	
$keep_g?$	cons g	
$keep_h?$	cons h	

chain pass

null

halt

There's a few new things to note here, as it's our first real source code using this style of grammar. For starters, we have a two new keywords: **induct** and **define**.

The `induct` operator declares this source code to be a function defining operator, yet as this is a generic keyword we still need to specify ⟨*induct_name*⟩ so we can reference the operator itself later on. A simple example of such a name might be "signature_maker". Following this, the `define` keyword reaffirms that we're defining a function, which itself needs to be followed by its *func_name* and any argument variables it accepts (in this case it accepts none).

Otherwise, the interpretation of the induction operator's body is that we set each *keep*? policy to its respective boolean value, and then run the ⋆{`open pass id`} `cpose` operator with `id` as the initial input. As for the column title **id**, I hadn't previously declared its meaning when I introduced the `distem cposes`.

Basically it just means each function in the column defaults to `id` so we don't need to specify it each time, and given that it's `id` each time it doesn't make a difference whether it's `call` or `pass`.

This model actually points out an interesting change in perspective: We seek to build tuples, but here we're not trying to build these tuples directly, we're trying to build the functions that build these tuples. Generally speaking, this is the underlying paradigm used to solve the signature and other function building problems. As a consequence of this strategy, we can now define an induction operator for our 32 flavors of `stem`:

```
induct stem_maker $stem\_name$ $keep_d?$ $keep_e?$ $keep_f?$ $keep_g?$ $keep_h?$ :

    define $stem\_name$ $policy?$ $d$ $e$ $f$ $g$ $h$

    chain pass dihold

    $policy?$
    open            pass            id
    $keep_d?$       cons $d$
    $keep_e?$       cons $e$
    $keep_f?$       cons $f$
    $keep_g?$       cons $g$
    $keep_h?$       cons $h$

    chain pass
    null
    $d$
    $e$
    transit $f$
    delay $g$ $h$

    halt
```

Notice in this version when defining the *policy*? function's signature we didn't start the `open` chain with `id` as its input like we had in the standalone version? The reason for this is we had more specific content available this time. I've mentioned this before, but by convention an `id` in these locations is just a placeholder signifying the end of a continuation passing line.

Also, note that we've now used the original `distem` operator and its `cpose`s to reimplement `stem`? What happens if we instead use `distem` it to reimplement itself? Such an implementation is said to be **meta-circular**, this terminology being borrowed from [4]. Such meta-circular operators tell us the bare minimum tools needed to rebuild themselves, which is an important theoretical boundary of our narrative design.[15]

Lattice Optimizations

There's actually one loose end within the previous example source code: The use of the $\star$\{`open pass id`\} cpose.

I had made a big fuss that primary modelling is better than secondary for actual implementation, pointing out that our particular secondary model had inefficiencies because it made unnecessary computations such as identity compositions. We then went on to accept the example primary model as better even though it does the same thing: The `id` modifier within $\star$\{`open pass id`\} defaults all functions passed to its column to be composed with the `id` function.

The reason this primary approach is still an improvement over the secondary is because we can systematically optimize these identity functions out of the final construction. This is due to the *operator lattice* that was previously used to introduce the core conditional composition operators:

[15]Not only that, but if we were to take an inventory of all the grammar used within a meta-circular construction, we would know the bare minimum tools needed from another language if we wanted to implement these tools from scratch. This has practical implications in library design, not to mention theoretical implications when it comes to consistency semantics.

$$\begin{tikzcd}[column sep=small]
 & & \text{distem}_{(3,3)} \arrow[dl] \arrow[dr] & & \\
 & \text{stem}_{(2,3)} \arrow[dl] \arrow[dr] & & \text{costem}_{(3,2)} \arrow[dl] \arrow[dr] & \\
\text{pend}_{(1,3)} \arrow[dr] & & \text{dihold}_{(2,2)} \arrow[dl] \arrow[dr] & & \text{copend}_{(3,1)} \arrow[dl] \\
 & \text{hold}_{(1,2)} \arrow[dr] & & \text{cohold}_{(2,1)} \arrow[dl] & \\
 & & \text{if}_{(1,1)} & &
\end{tikzcd}$$

Here I have presented it in its dual form to which we will now use in the following derivations.

The general approach to these optimizations is to first take our given `cpose` $\star$`{open pass id}`, and translate it into its `distem` form:[16]

$$\text{distem}(\,\textit{true}?,\ \textit{cont},\ -_{arg},\ \textit{next},\ \textit{cont},\ -_{arg},\ \text{id}\,)$$

The second step is to ignore the *true*? value at the beginning, and then strike out the `id` coordinate from the signature as it is not actually needed in `distem`'s computations:

$$\begin{array}{rcl}(\,\textit{true}?,\ \textit{cont},\ -_{arg},\ \textit{next},\ \textit{cont},\ -_{arg},\ \text{id}\,) & \longrightarrow & (\,\textit{cont},\ -_{arg},\ \textit{next},\ \textit{cont},\ -_{arg},\ \text{id}\,) \\ & \longrightarrow & (\,\textit{cont},\ -_{arg},\ \textit{next},\ \textit{cont},\ -_{arg}\,)\end{array}$$

We make note of the fact that `id` was located at the *sixth* coordinate of the tuple (when we ignore the boolean value *true*? ; or if we're indexing by zero). We now reference the above lattice to retrieve `distem`'s subscript:

$$(3,3)$$

The interpretation here is that this tuple represents the six arguments of `distem`'s signature (again ignoring the initial boolean value), and that this noted sixth argument belongs to the 3 arguments on the *right side* of the pair. Now, since we have removed the `id` argument from the tuple we should also update this information in our pair:

$$(3,2)$$

We use this derived index to reference the lattice again and finally arrive at the operator we will be using in our optimization: `costem`. We finish by combining this derived operator back with our derived signature (including the initial boolean value) to obtain $\star$`{open pass id}`'s optimized implementation:

$$\text{costem}(\,\textit{true}?,\ \textit{cont},\ -_{arg},\ \textit{next},\ \textit{cont},\ -_{arg}\,)$$

This general approach applies not only to `distem` and its cposes, it applies to any of the operators in the above lattice for which there is a meaningful downward arrow. Keep in mind though this only applies to the respective known `cpose`s when the derived operator also has its own cposes, otherwise the continuation passing process breaks down.

In anycase, this approach is so routine it could be standardized as its own induction operator—that is if we decided to use it in a *meta-circular* way to reimplement the `cpose`s of the various `stem`s. Either way, I have supplied a subset of these optimizations for reference in the appendix.

Refined Regular Induction

We left off the signature problem subsubsection nearly rederiving `stem` in a metacircular way. Given these optimizations, if we had at the time derived a "`costem_maker`" instead, it actually would have been metacircular.

Yet, these induction operators can do even more for us as they allow for alternative `stem` style operators where the *policy*? boolean value (as part of the signature) instead becomes a *boolean valued function*. With

[16]We chose the second modifier to also be a `pass`, but it could just as easily have been `call` without any loss of generality.

such induction operators we could even extend the various `cposes`, where their extensions also allow for the *policy*? variables to be boolean valued functions.

If we take this to its logical extreme, we can even extend *all* of our **regular induction** operators this way—not just our alternations. Keep in mind though, at this point such operators are potentially now outside the range of the theory of *regular languages*, and might require their own proofs as to things like the pumping lemma.

As for giving this *multiplicity* of new operators their own nomenclatures? I find overall that it is unnecessary. In practice the only extension I have (as of yet) ever used is the one where *policy*? accepts the $-_{arg}$ variable, which is to say the one passed from the previous step in the chain. As such, we can reuse the same names and notations and simply equip *policy*? with appropriate uses of the variable placeholder $-$ as needed, with no ambiguity ensuing.

To be safe though, in the cases where we want to be more overt about which signature variables the *policy*? function accepts, we might for example equip our notation as follows:

$$\textbf{open}_{\{1,4\}} \quad \textbf{call} \quad \textbf{pass}$$

meaning *policy*? accepts the first and fourth variables of its respective `stem` operator.

The Factorial Function

Let's now put all this theory to good use, and demonstrate an implementation of *the factorial function.*

I'm not going to assume the reader is fully comfortable with these newer notations just yet, so let's learn how to translate from the classical mathematical definition and its notation:

$$n! \quad := \quad \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{otherwise.} \end{cases}$$

The first step for us is to actually reimplement the factorial function with something slightly more general:

$$(p,n)! \quad := \quad \begin{cases} \ p & \text{if } n = 0, \\ (\,p \cdot n,\ n-1\,)! & \text{otherwise.} \end{cases}$$

In this case we can redefine our first factorial function as:

$$n! \quad := \quad (1,n)!$$

Next (and this step is a bit of a transition), we translate this *pair factorial* function into the `stem` operator style:

$$!\,p\ n \quad := \quad \text{stem}(\,\text{isZero?}\,n,\ \text{id},\ p,\ !\ (p\cdot n),\ \text{dec},\ n\,)$$

Here `isZero?` tests if n is zero, and `dec` just decrements it to become $n-1$. Also, notice that the right side recursive call of this pair factorial is a *curried* version:

$$!\ (p \cdot n)\ -$$

We now translate this further into its `cpose` form:

$$\begin{array}{lll} \textbf{closing} & \textbf{call} & \textbf{call}\ \ (p,n) \\ \text{isZero? cdr} & \text{car} & (\text{cons}\ (\cdot\ -)\ (\text{dec cdr}\ -)) \\ \text{isZero? cdr} & \text{car} & (\text{cons}\ (\cdot\ -)\ (\text{dec cdr}\ -)) \\ \text{isZero? cdr} & \text{car} & (\text{cons}\ (\cdot\ -)\ (\text{dec cdr}\ -)) \\ \quad\vdots & \ \vdots & \qquad\qquad\quad\vdots \\ \text{isZero? cdr} & \text{car} & (\text{cons}\ (\cdot\ -)\ (\text{dec cdr}\ -)) \end{array} \left.\vphantom{\begin{array}{l}a\\a\\a\\a\\a\end{array}}\right\} n+1 \text{ times}$$

Notice the traditional *policy*? value in the `closing` column is now a function? Hence, this is an extended version of our `closing` cpose where the input from the previous row is passed to the testing function as well. The "$n+1$ times" comment on the side is to say that we only need to repeat this chain so many times before we know it will halt, which is readily deducible from the factorial function itself. Also, I'm reusing the $(\cdot)$ function notation to multiply the contents of a pair:

$$\cdot\,(p, n) = p \cdot n$$

We're not quite done as this notation can be compressed using the `repeat` operator:

$$\begin{array}{lll}
\textbf{repeat}\ \ n+1 \quad (p,n) & & \\
\textbf{closing} & \textbf{call} & \textbf{call} \\
\text{isZero? cdr} & \text{car} & (\!(\ \cdot\ ,\ \text{dec cdr}\ )\!) \\
\textbf{closed} & &
\end{array}$$

In fact we've simplified further by using *bifunctions*. I haven't introduced this idea yet, and I don't want to go deeply into it because it takes us outside the scope of this essay, but we can extend our 1-dimensional functions to 2-dimensional (or higher), possessing identities such as:

$$\begin{array}{lcl}
f\,(\!(g_1,\ g_2)\!) & := & (\!(f \circ g_1,\ f \circ g_2)\!) \\
(\!(g_1,\ g_2)\!)\,f & := & (\!(g_1 \circ f,\ g_2 \circ f)\!) \\
(\!(f_1,\ f_2)\!)\ \bullet\ (\!(g_1,\ g_2)\!) & := & (\!(f_1 \circ g_1,\ f_2 \circ g_2)\!)
\end{array}$$

where we've used the $\bullet$ operator and the double parentheses $(\!(\,,\,)\!)$ to discern from 1-dimensional functions.

If instead we wanted an especially terse presentation, we can retranslate this into its horizontal format:

$$(p,n)\ [\ \text{car}\ \underset{\text{isZero? cdr}}{{}_\circ|_\circ}\ (\!(\ \cdot\ ,\ \text{dec cdr})\!)\ )^{n+1}$$

Notice the absence of the *read* direction? Here it is implied given the input pair (p,n) on the left-hand side.

Finally, we can then use this to define our original factorial:

$$n! \qquad := \qquad (1,n)\ [\ \text{car}\ \underset{\text{isZero? cdr}}{{}_\circ|_\circ}\ (\!(\ \cdot\ ,\ \text{dec cdr})\!)\ )^{n+1}$$

There's an important change in perspective worth noting in all of this: Up until now the construction and evaluation of regular functions we're independent of each other, but here this factorial function intermixes its construction with its evaluation, and the two can't be separated out. This is theoretically relevant as it might indicate the deeper nature of *recursion*, but it's also something we shouldn't take for granted when we're considering performance.

For example, in the cases where we *can* separate out construction from evaluation, we should. If we can construct the function first, independent of any input value, we can assign it a name with which to reference later on. Then any time we evaluate we only need refer to the already constructed function. On the other hand, if we intertwined construction with evaluation, then every time the function was called we'd be doing a lot of extra work for nothing as we'd be rebuilding it each time.

As for the factorial function here, if we wanted to minimize memory use in its construction we could instead reimplement it as the infinity repetition version:

$$n! \qquad := \qquad (1,n)\ [\ \text{car}\ \underset{\text{isZero? cdr}}{{}_\circ|_\circ}\ (\!(\ \cdot\ ,\ \text{dec cdr})\!)\ )^{\infty}$$

in which case we'd have to take a lazy policy when constructing it—we'd build only the next part of the function, evaluate what we could, then continue. In this case though, each individual `cpose` step is independent of the input, so we could at least seperate out the single `stem` operator representing those, so as to build it only once.

Context-Free Induction

The pumping lemma for regular languages was our turning point for regular induction as it demonstrated the memory limitations of this style of construction. Now that we have the methods of the signature problem at hand we're finally ready to change our focus to the next level of induction operators.

To help us ease into things let's actually return to discussing the limits of regular induction, but this time we'll keep things at an informal level: Such limits can be demonstrated by more closely inspecting any composition where a component function possesses an **arity** greater than 1. For example, let's consider what happens if we define some function h using `eq?` which itself takes two input. In particular, if we use the *string paradigm* of regular induction we can implement h as:

$$h \quad := \quad \text{eq?}\ f\ g$$

The problem here is that this is ambiguous. We can interpret such a definition as either of two possibilities:

$$h\ - \quad := \quad \text{eq?}\ (f\ -)\ (g\ -) \qquad\qquad \text{or} \qquad\qquad h\ - \quad := \quad \text{eq?}\ (f\ g\ -)\ -$$

which depending on f, g are entirely different functions. These two alternatives demonstrate that we can easily resolve such ambiguity ourselves by manually specifying parentheses, and this works fine in local contexts, but in the long run it is **context-free induction** that allows us to systematize this process, and mitigate this memory limitation overall.

Context-Free Grammars

We begin this narrative with an introduction to context-free languages and their grammars, for which I borrow heavily from [3]. Let's go straight to an example of a simple but well designed context-free grammar:

identifier	-	I	$\rightarrow$	$a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
factor	-	F	$\rightarrow$	$I \mid (E)$
term	-	T	$\rightarrow$	$F \mid T * F$
expression	-	E	$\rightarrow$	$T \mid E + T$

The idea of such a grammar is that it's made up of a collection of **productions**:

$$\{\ \mathcal{H}_\alpha \rightarrow \mathcal{B}_\beta\ \}_{\alpha,\beta}$$

which themselves are made up of **heads** $\mathcal{H}_\alpha$ pointing to their respective **bodies** $\mathcal{B}_\beta$. In practice, there are often many productions that share the same head but have different bodies, and so it is common convention to condense them into the appearance of a single production:

$$\mathcal{H} \rightarrow \mathcal{B}_0 \mid\ \ldots\ \mid \mathcal{B}_n$$

which in fact is what was done in the above example. Also, notice how such condensation aligns nicely with the *alternation* notation of regular induction? It's good to keep notations consistent when we can.

As for the components of the production bodies, we consider them to be *characters* which make up the strings of our context-free language. In particular, the characters that correspond to production heads are called **variables**, while those which don't are called **terminals**, and can otherwise be considered constants.

In our example grammar the variables are:

$$\{\ E, T, F, I\ \}$$

These variables have also been given natural language names, but this is only for semantic clarity for our sake—it is otherwise external to the grammar's actual definition. In anycase the terminals in this grammar are as follows:

$$\{\ a, b, 0, 1, (,), *, +\ \}$$

As for how these productions, variables, terminals are used within a context-free grammar: They are used to generate strings by means of what are called **derivations**. One specific variable is declared the **start** (for the whole grammar, not just the derivation), and from there we choose any of the `start`'s productions to take us to the next step by substituting its body. From there, if the given body contains variables of its own we further the derivation by substituting those with their own appropriately chosen productions. The derivation finally halts once we arrive at a string containing only terminals. That terminal string is our generated string.

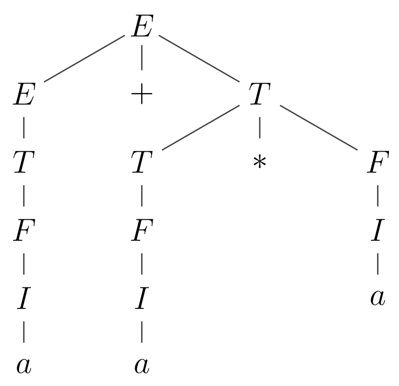
As for an example derivation using the above grammar:

$$\begin{array}{llllllllll} E & \Rightarrow & E+T & \Rightarrow & T+T & \Rightarrow & F+T & \Rightarrow & I+T \\ & \Rightarrow & a+T & \Rightarrow & a+T*F & \Rightarrow & a+F*F & \Rightarrow & a+I*F \\ & \Rightarrow & a+a*F & \Rightarrow & a+a*I & \Rightarrow & a+a*a & & \end{array}$$

Notice the use of the double bar arrows $\Rightarrow$ instead of the single bar ones $\rightarrow$ we used to define the productions? I doubt any confusion would arise if we instead used the latter $\rightarrow$, but to distinguish intent the former $\Rightarrow$ is standard. Also, I didn't specify it at the time, but our example grammar's start symbol is the variable E. As well, the intermediate strings that make up the steps of these derivations don't actually correspond to any predefined part of the context-free grammar, but for clarity they also have their own name: They are called *sentential forms.*

Next, given such derivations there is need to introduce the idea of their corresponding **parse trees**. For our example derivation we have its given parse tree as:

$$a + a * a$$



The top level of this tree is the start of the derivation, while each level of nodes below it represents the steps that follow.[17] Notice how each branch terminates in a leaf node which otherwise contains a terminal for its label? If we now parse through only these leaf node terminals (regardless of level) and collect them into a string, it represents the final result of the derivation.

Parse trees bring up one other consideration: *order of derivation.* So far we've only seen one example derivation which in fact was patterned as a *leftmost derivation.* Such derivations make their substitutions one at a time from the leftmost variable in each step. There are also *rightmost derivations* which always substitute from the rightmost variable instead. Then there's everything in between, which starts to get complicated. For our purposes we will stick to a leftmost policy.

This concludes the basic introduction to context-free grammars. It is relatively trivial to interpret their derivations as the induction operators we seek: We can use the production patterns to create the strings that become our chain compositions. With that said, it is less straightforward when we try and consider exactly how to translate such *manual* derivations into automated constructors. For that we will need to review the next level of automata.

Pushdown Automata

We review pushdown automata here because they are equipotent to context-free languages. This is to say that for every context-free grammar used to generate a string, it has a corresponding pushdown automata able to recognize it.

[17]This representation points out that our chain derivation although in appearance is sequential, is in fact only partially so: It defaults to being concurrent when the respective subderivations are independent.

Pushdown automata are *acceptors* with a similar nature to that of their *regular automata* predecessors. In fact they can be considered extensions of such finite state machines, but with an upgrade: They have a memory **stack**. A theoretical stack is a memory device with unlimited capacity to store information, but it's limited in how we can access it. Its interface can be summarized as *last in, first out.* This is to say the last thing we asked it to remember is the first thing we can access from it. If we remove that from its memory only then we can access the next last rememberance, and so on and so on.

Going back to our example context-free derivation,

$$\begin{array}{lllllllll} E & \Rightarrow & E+T & \Rightarrow & T+T & \Rightarrow & F+T & \Rightarrow & I+T \\ & \Rightarrow & a+T & \Rightarrow & a+T*F & \Rightarrow & a+F*F & \Rightarrow & a+I*F \\ & \Rightarrow & a+a*F & \Rightarrow & a+a*I & \Rightarrow & a+a*a & & \end{array}$$

how would a corresponding pushdown automata go about recognizing its derived string "$a+a*a$" ?

As it turns out, pushdown automata use a tried-and-true algorithm known as: Guessing and checking. To ease this process—at least conceptually—the version of automata we will use here is called *nondeterministic* pushdown automata with *ϵ-transitions.* This is just fancy speak for guessing and checking *in parallel*, where ϵ is the empty string and an ϵ-transition is a technicality that lets us divide and express such automata during their respective implementations. I offer these formal names as a reference for anyone who wants to learn more about the subject.

In anycase, let's now change our perspective to that of the pushdown automata itself: The way this parallel guessing and checking works is we're first given the string $a+a*a$ in its final form, but we're not told how it was derived. We then parse this string using a left-to-right policy, which also means we start from the leftmost character a. We seek to guess and check the first production of this derivation.

We now ask our given stack what its first symbol is, which by convention defaults to the *start symbol* E, for which we then scan all of the E productions and try each one. It's unlikely a single production will derive our full string, but at this point we only seek the productions which don't produce an immediate contradiction when tested against the current string. At this stage there are only two known productions for our start symbol:

$$\begin{array}{lll} E & \rightarrow & T \\ E & \rightarrow & E+T \end{array}$$

It's not immediately clear if either path will lead to a contradiction, but we don't go actively searching for possible contradictions: If they're going to find us, they're going to find us. As such, this is where we begin to apply the parallel guessing and checking algorithm: We test both paths as if we were running both processes independently and concurrently.

For the sake of convenience let's pretend we have a separate stack for each new thread we run. In this case for each path we replace our starting head E with its respective body:

$$\begin{array}{llll} \textbf{stack} & & \textbf{stack}_1 & \textbf{stack}_2 \\ E & \longrightarrow & T & E+T \end{array}$$

Keep in mind this is just a visual representation of our stacks—for convenience we can peer into them and read their whole content, but in actuality we only have access to the top of each stack, which we interpret here as their leftmost symbols.

Since we have not reached an indication of any contradictions, our second step is to now inspect the top symbol in each of the updated stacks, which in our example are T and E, respectively. We apply the same guess and check strategy and again replace these heads with the bodies of their respective productions:

$$\begin{array}{llll} \textbf{stack}_1 & & \textbf{stack}_{1,1} & \textbf{stack}_{1,2} \\ T & \longrightarrow & F & T*F \\ \\ \textbf{stack}_2 & & \textbf{stack}_{2,1} & \textbf{stack}_{2,2} \\ E+T & \longrightarrow & T+T & E+T+T \end{array}$$

In this case, we now have enough information to know that some of these threads will eventually lead to contradictions. For example $\texttt{stack}_{2,2}$ has two plus signs $+$ in its string, but the string we're testing against $a + a * a$ does not, so no matter how we continue this thread the final derivation will inevitably not match. Also, $\texttt{stack}_{1,2}$ contradicts. It's not as immediately obvious because our test string has a multiplication symbol $*$, but if you go back to the rules of the grammar:

identifier	-	I	$\rightarrow$	$a \mid b \mid Ia \mid Ib \mid I0 \mid I1$
factor	-	F	$\rightarrow$	$I \mid (E)$
term	-	T	$\rightarrow$	$F \mid T * F$
expression	-	E	$\rightarrow$	$T \mid E + T$

there is no longer any way to include a plus sign $+$ in its further derivations. In that case we can now rule out this possible thread as well, leaving us to continue with a reduced number of stacks:

$$\begin{array}{lcl}
\textbf{stack}_1 & & \textbf{stack}_{1,1} \\
T & \longrightarrow & F \\
\textbf{stack}_2 & & \textbf{stack}_{2,1} \\
E + T & \longrightarrow & T + T
\end{array}$$

and if we repeat our guess and check game yet again we have:

$$\begin{array}{lcll}
\textbf{stack}_{1,1} & & \textbf{stack}_{1,1,1} & \textbf{stack}_{1,1,2} \\
F & \longrightarrow & I & (E) \\
\textbf{stack}_{2,1} & & \textbf{stack}_{2,1,1} & \textbf{stack}_{2,1,2} \\
T + T & \longrightarrow & F + T & T * F + T
\end{array}$$

Here we also arrive at two new contradictions to rule out, where one of them is because our test string doesn't contain the left parenthesis symbol '(' and the other is because our test string in fact does contain the symbols $*, +$, but it does so in the opposite order than what's on the stack.

Applying this process once more we arrive at something different:

$$\begin{array}{lclll}
\textbf{stack}_{1,1,1} & & \textbf{stack}_{1,1,1,1} & \textbf{stack}_{1,1,1,2} & \ldots \\
I & \longrightarrow & a & b & \ldots \\
\textbf{stack}_{2,1,1} & & \textbf{stack}_{2,1,1,1} & \textbf{stack}_{2,1,2,2} & \\
F + T & \longrightarrow & I + T & (E) + T &
\end{array}$$

So far the front of our stacks have only had variables, but now we have some terminal symbols—in particular a, b. The $\texttt{stack}_{1,1,1,1}$ matches the first character of our string, but that's as far as it can go as there are no more variables on this particular stack to continue the process. As a consequence of this we have arrived at another contradictory thread, which is followed by two others when we explore the logic.

The only thread in the above which isn't contradictory is $\texttt{stack}_{2,1,1,1}$ which is what leads us to the first meaningful continuation in this whole process:

$$\begin{array}{lclll}
\textbf{stack}_{2,1,1,1} & & \textbf{stack}_{2,1,1,1,1} & \textbf{stack}_{2,1,1,1,2} & \ldots \\
I + T & \longrightarrow & a + T & b + T & \ldots
\end{array}$$

Here $\texttt{stack}_{2,1,1,1,1}$ matches the first two characters of our test string $a + a * a$, and even has a variable to continue the thread. As such, we've now reached the end of round one of this recursive loop. We pop the front terminals from both the stack and our string:

$$a + T \quad \text{becomes} \quad T \qquad \text{and} \qquad a + a * a \quad \text{becomes} \quad a * a$$

Then we start over again with the reduced string $a * a$ and the reduced stack:

$$\begin{array}{l} \textbf{stack}_{\text{round 2}} \\ T \end{array}$$

and that's the general algorithm! If we end up arriving at contradictions for all the parallel threads we had started, it means the test string does not in fact belong to our language of interest. Otherwise, if we successfully find a derivation then we have confirmed that the string in question does in fact belong.

Context-Free Bind Operators

Pushdown automata are great, but they weren't our final goal either. How can we use the pushdown algorithm just learned to derive strings rather than to recognize them?

For our example string "$a + a * a$" we would like to be able to model our induction grammar based on the expressive form of its context-free derivation:

$$\begin{array}{llllllll} E & \Rightarrow & E+T & \Rightarrow & T+T & \Rightarrow & F+T & \Rightarrow & I+T \\ & \Rightarrow & a+T & \Rightarrow & a+T*F & \Rightarrow & a+F*F & \Rightarrow & a+I*F \\ & \Rightarrow & a+a*F & \Rightarrow & a+a*I & \Rightarrow & a+a*a & & \end{array}$$

Truth be told this exact form isn't the most user-friendly for our purposes. If we're reading over this derivation in person it's a bit awkward to follow the logic without having to first memorize the productions for this particular grammar. Such memory constraints don't generally scale given that there are infinitely many context-free grammars in existence, and we certainly could not memorize the productions for them all.

For me the most straightforward and scalable way to express context-free derivations is to use their productions directly:

	production	//	**assumes**	**derives**
1)	$E \to E+T$	//	E	$E+T$
2)	$E \to T$	//	$E+T$	$T+T$
3)	$T \to F$	//	$T+T$	$F+T$
4)	$F \to I$	//	$F+T$	$I+T$
5)	$I \to a$	//	$I+T$	$a+T$
6)	$T \to T*F$	//	$a+T$	$a+T*F$
7)	$T \to F$	//	$a+T*F$	$a+F*F$
8)	$F \to I$	//	$a+F*F$	$a+I*F$
9)	$I \to a$	//	$a+I*F$	$a+a*F$
10)	$F \to I$	//	$a+a*F$	$a+a*I$
11)	$I \to a$	//	$a+a*I$	$a+a*a$
	terminal			

For each step in this new interface (they are numbered for reference), we show the production used to create it. This is the **production** column. As for the two side columns, they are commented out because technically they don't contribute anything to the derivation itself. They're only there for our benefit—so we can follow through the derivation as if it were a human readable *proof*.[18]

Let's go through how such a derivation would work, but this time focusing on what's going on behind the

[18]Such grammatical constructs could be extended in practice to formally include these side columns. As stated, they wouldn't contribute anything to the derivation itself, but the implementation of such grammars could then test these additional considerations against various logics so as to validate, confirm, and verify.

scenes. To do that, we change this *production oriented* interface just a little bit further still:

	production	**stack**	**composite call**
0)		E	id
1)	$E \to E + T$	$E + T$	id
2)	$E \to T$	$T + T$	id
3)	$T \to F$	$F + T$	id
4)	$F \to I$	$I + T$	id
5)	$I \to a$	T	$a+$
6)	$T \to T * F$	$T * F$	$a+$
7)	$T \to F$	$F * F$	$a+$
8)	$F \to I$	$I * F$	$a+$
9)	$I \to a$	F	$a + a*$
10)	$F \to I$	I	$a + a*$
11)	$I \to a$	$\varnothing$	$a + a * a$
	terminal		

As with our previous induction grammars, we're effectively building a function by passing it along the **composite**, but this time around we also equip our grammar with a **stack** construct similar to that of pushdown automata. The `composite` here is initialized with its default placeholder `id`, while the `stack` is initialized to contain only the *start symbol* E.

We then update both the `stack` and the `composite` in line 0 by using the `production` of line 1. The `stack` in this line is now the result of substituting the existing top element with the given production body. We effectively follow the same pattern as the pushdown automata algorithm, continuing this process as we move down the rows, until we get to line 5.

In this step, when we make the necessary substitution to update the `stack` we end up with the sentential form $a + T$, which differs from previous steps because we now have a terminal at its top. We continue shadowing the pushdown algorithm, except here instead of simply popping this terminal from the `stack` we also move it over to the `composite` column.

As a point of fact we've appended the terminal to the back of the current function. As such, I have labelled the column with a **call** modifier. Notice how the plus sign + terminal has also been moved from the `stack` over to the `composite`? Once we start the process of moving terminals to the composition it's necessary to move them all to continue. In summary of line 5, we apply the `production` and then update the `stack` to T, while the `composite` then becomes $a+$. It is at this point that the algorithm repeats—that is until the `stack` becomes empty, at which point it halts.

This is the overall idea of our context-free induction grammar, except such an interpretation still doesn't lend itself directly to the implementation we seek. In that case, we turn to the idea of a `bind` operator, the conventional wisdom being that wherever there's a *function* to be evaluated, there's a `bind` operator to evaluate it. This also suggests that wherever there's a function to be constructed, there's a bind operator for that as well.

This inspires the following `bind`, which we call **derive**:

```
define  (stack, composite)  ⊢{derive}  (head, body)

repeat  ∞  (stack, composite)
closing                                   pose                     chain call

alias  (local_stack, local_composite)
assign  (front, rest)  local_stack

eq? front  head                           chain pass               cdr
                                          cons                     push front
                                          concat body              cons rest
                                          rest local_composite
end
```

The intention of this algorithm is to take a (*stack*, *composite*) pair, and apply the *production* which here is represented as (*head*, *body*). The expected return is the next *stack* and *composite* pair in the derivation.

There's actually a lot going on in this source code, so we should go over the details. For starters, if you're not use to *abstract* bind operators you might take issue with the (*head*, *body*) pair representing a function. Referring back to bind's notational definitions we have:

$$\begin{aligned} x \vdash f &= f \dashv x \\ &:= f\langle x \rangle \end{aligned}$$

It seems implicit that f should be a function. Remember, our theory of binds comes from category theory, but this isn't the way category theory actually works: As a matter of fact anything can be a *morphism* with respect to how a given category is interpreted—not just the intuitive functions that are a natural inspiration. If this isn't fully satisfactory keep in mind that the (*head*, *body*) pair is meant to represent a *production*, which if it's more to your liking we could reinterpret to be a function if we really needed to.

Next, notice the use of the `alias` and `assign` operators? These grammatical constructs haven't been introduced yet. The `alias` keyword is more of a convenience than anything, it allows us to *pattern match* the current input and give temporary names to its internal structure. This is just to help clarify the situation for the human code writer. As for `assign`, it goes back to the methodology of *compression*, and the idea of *scope signatures*: In effect we are *assigning* a variable to a value so that we only need to compute it once, and can otherwise refer to it from then on. For our convenience it can also pattern match.

Lastly, I should state to be fair that this particular implementation doesn't following best practice coding, though you might not realize it as this grammatical form is still new to us. In particular notice the internal column functions:

chain pass	**chain call**
cons	cdr
concat *body*	push *front*
rest local_composite	cons *rest*

It would have been better to have defined these as their own functions in advance given that they tend to read as a bit awkward here and otherwise obscure the intended meaning. In fact we could have done as much using `assign`, but it didn't exist for us just yet.

Beyond that, note that this above `bind` operator implementation is rather streamlined: There's a lot of things that could in fact go wrong which would break the consistency semantics. The idea is that instead, if everything went right—meaning all the necessary assumptions were met—this algorithm would work as advertised. Let's now examine what could go wrong:

- We assumed the stack was not empty, but it might be. This assumption is evident by the fact that we pattern matched against its first and second elements with the `assign` operator.
- We assumed the production was valid for the underlying context-free grammar, and it might not be.
- We assumed if the *front* of the stack did not equal the *head*, it meant the *front* was a terminal. It's possible the production is valid, but the derivation as a whole isn't. In which case both the *front* and *head* might be variables which are simply not equal.

With these considerations in mind, we can now reinforce the stability of this code with several tests to make sure the assumptions are actually valid before we apply the bind:

```
induct derivation_maker $context\_free\_grammar$ :
    assign $cfg$ $context\_free\_grammar$
    define $(stack,\ composite)$ $\underset{cfg}{\Rightarrow\{\text{derive}\}}$ $(head,\ body)$
    $(head,\ body)$
    closing                              pose          id
    isNotProduction? $cfg$               $error\ 0$              // if it's not a production,
                                                                 // break with $error$ 0,
                                                                 // otherwise continue.
    isNull? $stack$                      $error\ 1$              // if the stack is empty,
                                                                 // break with $error$ 1,
                                                                 // otherwise continue.
    assign $front$ car $stack$
    isVariable? $cfg\ front$                                     // if the front of the stack is
    and not eq? $front\ head$            $error\ 2$              // a variable, but not equal
                                                                 // to the head, then break
                                                                 // with $error$ 2, else...
    chain call
    $(stack,\ composite)$ $\vdash\{\texttt{derive}\}$            // apply $\vdash\{\texttt{derive}\}$
    halt
```

With this algorithm at hand we are almost ready to translate the previous vertical *derivation* grammatical form into its proper implementation, but for one additional function definition, the **start** of the bind:

$$\text{start}(E) \quad := \quad (E,\ \text{id})$$

Finally, our example derivation with its $a + a * a$ construction can be expressed as follows:

	production				
0)	$\text{start}(E)$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$	6)	$T \to T * F$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$
1)	$E \to E + T$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$	7)	$T \to F$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$
2)	$E \to T$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$	8)	$F \to I$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$
3)	$T \to F$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$	9)	$I \to a$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$
4)	$F \to I$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$	10)	$F \to I$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$
5)	$I \to a$	$\underset{\text{cfg}}{\Rightarrow\{\text{derive}\}}$	11)	$I \to a$	
	terminal				

we then retrieve our final composition from the second component of the returned pair.

Admittedly in the case of context-free induction, for this to actually work we would have to specially interpret terminals such as the left '(' and right ')' parentheses, not to mention the use of *infix* functions such as $+, *$. Actually, regarding infix operators, so far we've taken them for granted so just to be thorough here I will specify their rule of application:

$$-_x\ \{infix\}\ -_y \quad = \quad \{prefix\}\ -_x\ -_y$$

Ambiguity

One aspect of context-free grammars we have not yet discussed is the idea of **ambiguity**.

In particular, ambiguity is an issue that initially arises with pushdown automata: In the process of trying to recognize a string the automata will determine if the string belongs or not, but in the case that it does nothing is said about whether the parse tree associated with its derivation is unique.

This wasn't an issue with (what is at this point) our canonical string example $a+a*a$, as the productions of its context-free grammar were designed to prevent such ambiguity, but if for example we condense the underlying grammar to the following:

$$\begin{array}{lclcl} \text{identifier} & - & I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ \text{expression} & - & E & \rightarrow & I \mid E+E \mid E*E \mid (E) \end{array}$$

we are still able to derive the string $a+a*a$, but now there are alternative paths to do so:

left derivation:

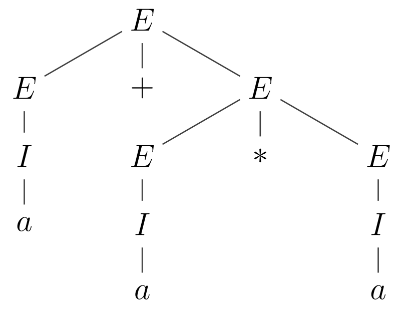
$$\begin{array}{lllllllll} E & \Rightarrow & E+E & \Rightarrow & I+E & \Rightarrow & a+E & \Rightarrow & a+E*E \\ & \Rightarrow & a+I*E & \Rightarrow & a+a*E & \Rightarrow & a+a*I & \Rightarrow & a+a*a \end{array}$$

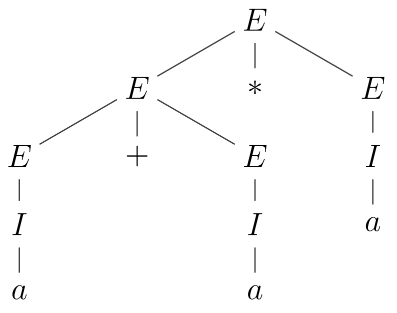
right derivation:

$$\begin{array}{lllllllll} E & \Rightarrow & E*E & \Rightarrow & E*I & \Rightarrow & E*a & \Rightarrow & E+E*a \\ & \Rightarrow & E+I*a & \Rightarrow & E+a*a & \Rightarrow & I+a*a & \Rightarrow & a+a*a \end{array}$$

To be clear, ambiguity isn't an issue because there exists more than one derivation for a string,[19] it is an issue solely due to the existence of more than one parse tree:

$$a+a*a$$





Any corresponding pushdown automata could tell us which one it used in *its* derivation, but not which one was used in the original.

The problem here is that parse trees implicitly represent the means by which we *group* the terms of a string. Returning to our canonical string, its terms can be grouped as $a+(a*a)$ or $(a+a)*a$, but with respect to their intended meanings they are generally different:

$$a+(a*a) \quad \neq \quad (a+a)*a \qquad\qquad \text{// sometimes equal, but not usually.}$$

This also ties back to where we started in context-free grammar theory, with our ambiguous and *motivating* composition:

$$h \quad := \quad \text{eq? } f \ g$$

which as we remember could be interpreted to represent two distinct functions:

$$h\ - \quad := \quad \text{eq? } (f\ -)\ (g\ -) \qquad\qquad \text{or} \qquad\qquad h\ - \quad := \quad \text{eq? } (f\ g\ -)\ -$$

[19]Keep in mind it's possible to have more than one derivation for a string such that they still result in the same parse tree.

Ambiguity in grammars is an issue for us then because parse trees are inevitably what determine the order of evaluation of the functions we inductively construct. We may be able to solve the problem locally with parentheses, but it's not something we should simply assume as given within the larger discourse.

As for the nature of ambiguity within this larger discourse? It is in fact relative to the *context-free languages* that such grammars represent. Some languages are **inherently ambiguous**, this being in the sense that all grammars that generate them are themselves ambiguous. Unfortunately there's no easy way to differentiate these various kinds of languages: It is a known result that there is no single universal algorithm to test for ambiguity of a language.

In terms of languages which aren't inherently ambiguous, by logical necessity at least one of their respective grammars is safe for our evaluations, but it's certainly possible other grammars which generate the same language aren't. As there is no universal algorithm to test languages, it also means there's no universal algorithm to translate an unsafe grammar (of an otherwise unambiguous language) into a safer form.

All of this is to say any logical verifications or mitigating translations are a case by case, one ad-hoc proof at a time, sort of situation. If we're especially clever we might find *kinds* of context-free grammars which do share proofs or translations, but that's just about as good as it gets. In anycase, this is an important issue to recognize and accept as once we acknowledge such complexity we can begin to mitigate it. With that said, are there at least some best practices?

Yes. Generally speaking the first best approach when we can't reduce the complexity of a context-free language is to reduce the complexity of the grammar we use as its interface. We do this by translating its productions into an alternative set which are simpler in form but still generate the same language. There are four specific strategies:

1. We can reduce all ϵ-productions down to the single default start symbol production $S \to \epsilon$.

2. We can get rid of the *unit productions*, which are the rules that cast one variable into another $U \to V$.

3. We can get rid of what are known as *useless symbols*, which are otherwise present in some grammars but don't actually contribute to the derivations of any terminal strings.

4. We can translate a given grammar into what's called the **Chomsky normal form** where the only bodies for their respective productions are either:

   (a) single terminals, for example $H \to t$ for variable H and terminal t.

   (b) two catenated variables, for example $H \to UV$ for variables H, U, V.

To be clear, these approaches alone don't mitigate ambiguity, rather they are the first steps toward creating **tiebreaking** strategies to disambiguate parse trees. Such clarified productions can then be used within single serve proofs that the given tiebreakers do in fact create unambiguous grammars. Unfortunately there's also a tradeoff: In practice the forms of individual productions may simplify, but in turn the total number of such productions often increases.

I should mention there are other applications of the above clarifying strategies, for example the Chomsky normal form in particular improves the performance of the parsing algorithms for its respective languages. In anycase, the implementation of such grammatical translations as well as any resulting parser optimizations are beyond the scope of this essay, for that I would refer you back to [3].

Finally, there is one additional method worth mentioning here to mitigate ambiguity: We *prevent* it from happening in the first place—for which our induction grammar is then ideal. By specifying the derivation of a function rather than just hand coding it, a compiler could create and store the parse tree as part of the definition. When the function shows up later in the source code, the parse tree is already known thus preventing the ambiguity in the first place.

The Context-Free Pumping Lemma

As with regular languages, context-free languages have their own version of the pumping lemma:

> Let $\mathcal{L}$ be a context-free language with strings $u, v, w, x, y, z \in \mathcal{L}$ such that $uvwxy = z$ and $vx \neq \epsilon$. There exists a natural number $n \in \mathbb{N}$ (dependent only on $\mathcal{L}$) such that if $\text{length}(z) \geq n$, and $\text{length}(vwx) \leq n$, then for all $k \in \mathbb{N}$ we have:
>
> $$uv^k wx^k y \in \mathcal{L}$$

The proof of this lemma is somewhat abstract, but reduced to its simplest idea it again rests on the *pigeonhole principle.* As broad outline of the proof we actually start with a grammar for the language in question and translate it into its *Chomsky normal form.* The parse trees end up translating into *binary trees* for which it then becomes easier to find a pigeonhole induced memory constraint—creating the stated repetition.

Given that this schema of *inducing repetition through the pigeonhole principle* shows up in both this and the regular pumping lemma, I suspect that wherever one meets a repeating pattern within automata—or maybe even in life's designs—there's probably some memory limitation hidden in its details that forces such symmetry in the first place. This is an interesting thought, though it's neither here nor there.

Regular Languages as Context-Free Grammars

As we're nearing the end of our *context-free* discussion, let's look more closely at the relationship between context-free induction and regular induction. In particular, context-free functions are an extension of regular functions, which can be seen from the result that context-free languages are extensions of regular ones.

To understand this, we first might recall that regular expressions are constructed from *catenation*, *alternation*, and *repetition.* As such, we only need describe these generic operators in a context-free way, as I *naively* do so here:[20]

$$\begin{array}{llcll} \text{characterization} & - & S & \to & \epsilon \mid c_1 \mid \ldots \mid c_n \\ \text{parenthesization} & - & P & \to & (E) \\ \text{repetition} & - & R & \to & E^* \\ \text{catenation} & - & C & \to & EE \\ \text{alternation} & - & A & \to & E \text{ `}|\text{' } E \\ \text{expression} & - & E & \to & S \mid P \mid R \mid C \mid A \end{array}$$

In this grammar E is the start symbol. Note the single quotes '|' enclosing the alternation bar? They're there to distinguish between the bar symbol used within regular expressions, and the ones used in context-free productions.

As example of this regular grammar, we can now derive the following expression:

$$\begin{array}{llllllll} E & \Rightarrow & C & \Rightarrow & EE & \Rightarrow & PE & \Rightarrow & (E)E \\ & \Rightarrow & (A)E & \Rightarrow & (E|E)E & \Rightarrow & (S|E)E & \Rightarrow & (c_1|E)E \\ & \Rightarrow & (c_1|S)E & \Rightarrow & (c_1|c_2)E & \Rightarrow & (c_1|c_2)R & \Rightarrow & (c_1|c_2)E^* \\ & \Rightarrow & (c_1|c_2)S^* & \Rightarrow & (c_1|c_2)c_3^* \end{array}$$

with c_1, c_2, c_3 being characters belonging to the underlying regular alphabet.

There is actually a second way to interpret regular expressions as context-free grammars: Each fully defined expression was originally meant to represent its own language of strings, and so could also be translated into its own set of context-free productions representing that same language.

Let's break down and translate the major components of our current derivation $(c_1|c_2)c_3^*$. We start by observing this expression has a repetition c_3^*, so we first translate it into the following productions:

$$R \to Rc_3 \mid \epsilon$$

This works because we can keep substituting R in the body until we've repeated the character c_3 the number of times we wanted, then we can close it off with ϵ. Our expression also has a catenation, which we translate as:

$$C \to AR$$

Finally, our expression has an alternation $(c_1|\,c_2)$ for we can define the alternatives as individual productions:

$$A \to c_1 \mid c_2$$

[20]I describe this translation as naive because it would need to be proven to generate exactly the regular expressions with alphabet $\{c_1, \ldots, c_n\}$. As this is an essay only meant to convey major ideas, I do not confirm this result here, and although I do not want to set a bad example I must acknowledge I have not in fact formally verified this grammar myself. My reasoning is that this particular grammar translates the recursive definition of regular expressions quite faithfully, and if nothing else at least contains the language of regular expressions, which for the purpose of this subsubsection is more relevant.

If we now define our start symbol as C, this grammar can be summarized *freely* as:

$$\begin{array}{llccl} \text{repetition} & - & R & \rightarrow & Rc_3 \mid \epsilon \\ \text{alternation} & - & A & \rightarrow & c_1 \mid c_2 \\ \text{catenation} & - & C & \rightarrow & AR \end{array}$$

which will generate the language matching the expression $(c_1|c_2)c_3^*$.

Any other regular expressions can be translated to their respective context-free productions in similar ways. In anycase, this outlines how regular induction operators can be reimplemented as context-free derivations.

Grammatical Paths and Context-Free Grammars

We haven't gotten much into the function semantics of [2], but there does seem to be an overlap between grammatical path notation and context-free parse trees—both using trees to represent functions. Overall, untyped grammatical path functions are universally computable and are thus the more general theory, but their tree oriented design does still suggest a connection or two. It's beyond the scope of this essay to explore these in any serious way, but I would still like to present the basic correspondences.

For starters, it's relatively easy to show that the definitions of the *bodies* and *signatures* of grammatical path functions are derivable using context-free grammars. I won't formalize anything here,[21] but the bodies of grammatical paths always start with a root function for which we then build more complex bodies by substituting additional functions for their argument variables (creating compositions). This could be represented as $V \rightarrow B$. Otherwise, we substitute actual argument values, which could be represented as $V \rightarrow a$. For any of this to work though, we'd need a set of predefined function primitives, but instead of giving them their own productions (such as $F \rightarrow f$) it would be better to embed them directly into body productions such as:

$$B \quad \rightarrow \quad f\,\underbrace{V\,\ldots\,V}_{arity}$$

The assumption here is that any given function primitive will have a fixed **arity** which we could then tailor for its respective production. The productions of signatures could be theorized in a similar way.

On the flip side, showing a correspondence in the other direction—where we define context-free derivations using grammatical paths—is actually a bit trickier. To walk us through it, we'll once again refer to our former example grammar:

$$\begin{array}{llccl} \text{identifier} & - & I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\ \text{factor} & - & F & \rightarrow & I \mid (E) \\ \text{term} & - & T & \rightarrow & F \mid T * F \\ \text{expression} & - & E & \rightarrow & T \mid E + T \end{array}$$

along with its example derivation:

$$\begin{array}{lllllllll} E & \Rightarrow & E+T & \Rightarrow & T+T & \Rightarrow & F+T & \Rightarrow & I+T \\ & \Rightarrow & a+T & \Rightarrow & a+T*F & \Rightarrow & a+F*F & \Rightarrow & a+I*F \\ & \Rightarrow & a+a*F & \Rightarrow & a+a*I & \Rightarrow & a+a*a & & \end{array}$$

The first step in specifying context-free grammars as grammatical paths is to translate their productions into something more along the lines of function definitions:

$$\begin{array}{llccl} \text{identifier} & - & I(index) & := & \underset{index}{a} \mid b \mid I(-)a \mid I(-)b \mid I(-)0 \mid I(-)1 \\ \text{factor} & - & F(index) & := & \underset{index}{I(-)} \mid (E(-)) \\ \text{term} & - & T(index) & := & \underset{index}{F(-)} \mid T(-_1) * F(-_2) \\ \text{expression} & - & E(index) & := & \underset{index}{T(-)} \mid E(-_1) + T(-_2) \end{array}$$

[21]If it were to be done, one potential advantage in formalizing a context-free grammar for grammatical path components is that the derivations would actually suggest how such functions could be evaluated, though a few tweaks would be needed to make recursion work.

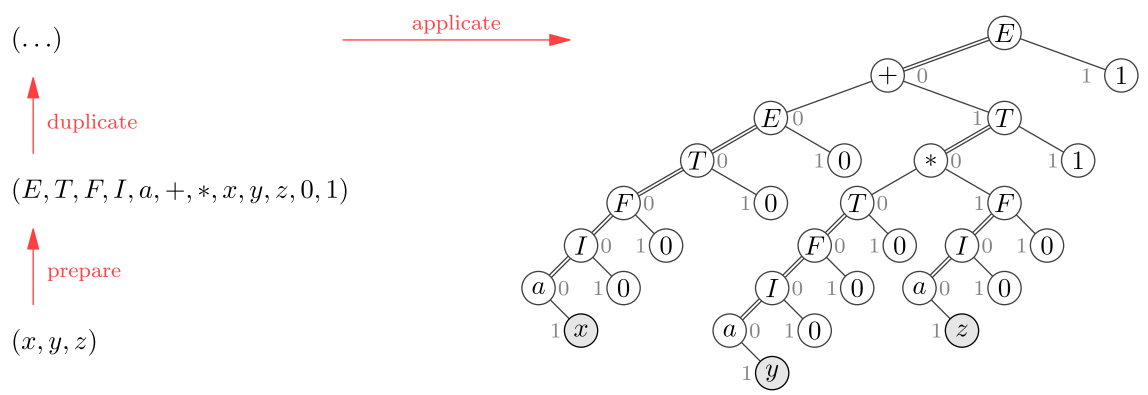
Here we're also using the regular induction paradigm for our convenience.

As for our example derivation, it would then translate as follows:

$$\begin{array}{llllll}
E(1) & \Rightarrow E(0)+T(-) & \Rightarrow & T(0)+T(-) & \Rightarrow & F(0)+T(-) \\
 & \Rightarrow I(0)+T(-) & \Rightarrow & a(-_x)+T(1) & \Rightarrow & a(-_x)+T(0)*F(-) \\
 & \Rightarrow a(-_x)+F(0)*F(-) & \Rightarrow & a(-_x)+I(0)*F(-) & \Rightarrow & a(-_x)+a(-_y)*F(0) \\
 & \Rightarrow a(-_x)+a(-_y)*I(0) & \Rightarrow & a(-_x)+a(-_y)*a(-_z) & &
\end{array}$$

Finally, if we were to translate this into a grammatical path equivalent, we would have:

$$a(x)+a(y)*a(z)$$



The signature used in the `applicate` mapping is shortformed to $(\ldots)$ for clarity, as it is otherwise quite large given the bijective nature of the mapping.

This demonstrated correspondence is actually quite intriguing: From a function semantics perspective, it suggests context-free grammars are the logical end result of **currying**. Admittedly I did not make such a connection myself until I translated this derivation, but it makes sense actually: From a function semantics perspective both recursion and *currying* are the first natural extensions of composition, and so any induction paradigm based off of composition would inevitably lead to their use as well.

Context-Sensitive Induction

We are finally ready to move beyond context-free induction. If we were to continue following the conventional narrative from automata theory, the next level of languages to explore would be **context-sensitive languages** with their corresponding **linear bounded automata**.

For our purposes this is unnecessary. In the traditional theory linear bounded automata could be considered restricted forms of Turing machines: They are *bounded linearly* in the amount of memory available to them relative to the memory needed for a given string input—and are otherwise built the same way. This means anything that can be derived as true for Turing machines also holds for linear bounded automata with enough memory. This makes their pedagogical exposition mostly redundant.

To be fair, linear bounded automata are actually better approximations to the hardware implementations of Turing machines, but we will admittedly overlook them here regardless and immediately move on to the next level of induction.

Register Induction

We are now at the highest level of languages for our induction operators.

Turing Machines

This level of languages is primarily represented by what are known as *Turing machines*. These devices can still be viewed as *finite state machines*, except here the finite state is the set of *instructions* (source code) for the given machine. Otherwise, the machine has access to a potentially infinite supply of memory similar to that of pushdown automata—but unlike such automata, Turing machine memory isn't a stack, it is *random access*.

Turing machines are theoretically important for two major reasons: 1) They are known to be equipotent to the class of *computable functions* defined in mathematics—specifically μ-recursive functions. 2) As far as equipotent machines go, they have a fairly intuitive and simple design which makes them ideal for mathematical proofs.

In particular, one of the foundational theorems of computing science is what's known as **the halting problem**: If we were given the *instruction set* for any specific Turing machine, as well as some initial input (stored in its memory system), there is no single other Turing machine that could tell us if our specific machine would eventually halt.

This theorem tells us that Turing machines have a fundamental limitation, but unlike regular or context-free languages which also have limitations (due to their respective pumping lemmas), the halting problem isn't a limitation on which languages of strings can be verified, it is a limitation on the representation of computable functions themselves.

Another way to understand this is to relate it back to the primary and secondary modelling approaches of design. This result says that there's no primary *constructive* definition of a computable function, and that any such definition must be secondary. What this means is that Turing machines, or μ-recursive functions are objects which form some primary model that properly contain the *well behaved* computable functions. These in turn have to be submodelled as the subspace of machines which actually do halt.

As a consequence, we do have a universal grammar, but we have to accept that some of the *expressions* we create with it might not be the idealized computable functions we seek. What's more: There's no universal way to prove such expressions as one or the other. It's ad-hoc, one at a time.

Register Machines

Although Turing machines are theoretically important they are also less practical than *register machines*—which are a known equivalent. It is for this reason we will focus on determining register induction grammars rather than Turing ones.

We can think of register machines as consisting of (potentially infinite) collections of *registers*, along with a *stack* for convenience. Each machine is described by an instruction set called its **controller**. For our induction purposes we can interpret each register as a memory cell containing a single function, though we're getting ahead of ourselves at this point.

Since we seek a grammar for register induction, let's start by observing a dual grammar for register machines. The following is a well designed collection of instructions described in [4] using LISP notation for building such machines and their controllers:

(reg ⟨*register-name*⟩)

(const ⟨*constant-value*⟩)

(assign ⟨*register-name*⟩ (reg ⟨*register-name*⟩))

(assign ⟨*register-name*⟩ (const ⟨*constant-value*⟩))

(assign ⟨*register-name*⟩ (op ⟨*operation-name*⟩ ⟨$input_1$⟩ ... ⟨$input_n$⟩))

(assign ⟨*register-name*⟩ (label ⟨*label-name*⟩))

(test (op ⟨*operation-name*⟩) ⟨$input_1$⟩ ... ⟨$input_n$⟩)

(branch (label ⟨*label-name*⟩))

(goto (label ⟨*label-name*⟩))

(goto (reg ⟨*register-name*⟩))

(save ⟨*register-name*⟩)

(restore ⟨*register-name*⟩)

If this looks a lot like the grammar for machine languages it's because it pretty much is: It is an abstraction of the assembly languages one finds for actual hardware processors.

Such machine grammar is a good place to start, and it helps us obtain a better picture of what we will generally need for our own induction grammar, but it's also somewhat removed from the underlying endoposes we have been working with for our past induction operators.[22] With that in mind, we now turn to the final form of function induction grammars within this essay:

memory x			**composite** y		
closing	**call**	**call**	**open**	**pass**	**call**
$policy?_{\mathbf{m},0}$	$break_{\mathbf{m},0}$	$f_{\mathbf{m},0}$	$policy?_{\mathbf{c},0}$	$next_{\mathbf{c},0}$	$f_{\mathbf{c},0}$
$policy?_{\mathbf{m},1}$	$break_{\mathbf{m},1}$	$f_{\mathbf{m},1}$	$policy?_{\mathbf{c},1}$	$next_{\mathbf{c},1}$	$f_{\mathbf{c},1}$
$policy?_{\mathbf{m},2}$	$break_{\mathbf{m},2}$	$f_{\mathbf{m},2}$	$policy?_{\mathbf{c},2}$	$next_{\mathbf{c},2}$	$f_{\mathbf{c},2}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$policy?_{\mathbf{m},n}$	$break_{\mathbf{m},n}$	$f_{\mathbf{m},n}$	$policy?_{\mathbf{c},n}$	$next_{\mathbf{c},n}$	$f_{\mathbf{c},n}$
closed			**closed**		

In a lot of ways this grammatical form is quite straightforward: In appearance at least, it is a simple extension of our `distem cpose` grammar, just doubled up. That would be the obvious interpretation, but is in fact misleading, and for a few reasons.

For one, the parallel `distem`'s aren't independent here, they can actually communicate. This is to say they not only build functions, they are able to pass the functions they're implicitly building (or parts thereof) directly to each other. Aside from that, the other subtlety of this new grammatical form is the realization that we've now extended beyond `distem` itself, and in doing so we've possibly opened up new design issues.

For example if you go back to the definition of `distem` there's more than one way to extend it—leading to multiple alternative extensions all equally valid and potentially interesting. How do we decide which to use? What kinds of nomenclatures do we equip such inventories of operators with? Fortunately for us, such issues are *moot*: There's a reasonably well known automata theorem which says a finite state automata equipped with *two* stacks is equipotent to a Turing machine.

Lo and behold, this is effectively what we have when we use the above grammatical form: The **memory** is our intended *stack*, while the **composite** is a set of registers which are used not only to hold the function we're building, but which hold other objects as a secondary *stack*.

Note, this equates to *random access* memory specified within Turing machines: This works because with two stacks we could access arbitrary locations within either of the individual stacks by simply shifting everything before (the given location) onto the other stack, for which we could then read or write to that location, followed by shifting everything back. In theory this is even how the above grammatical form works, but in practice we would generally prefer to optimize toward random access performance.

Final note: I had previously introduced the `assign` and `alias` operators which allowed us to associate variables with given values. In theory the memory system used within this two column construct could be implemented as a reserved portion of the memory stack, simplifying this narrative design nicely as well.[23]

Register Binds

As for translating this grammatical form into its respective `cposes` (which could in turn be translated into proper monadic endopositions), their translations are largely straightforward, though here I will use notation I had previously introduced in the *factorial function* subsubsection:

$$\begin{array}{lcl} f \llparenthesis g_1,\ g_2 \rrparenthesis & := & \llparenthesis f \circ g_1,\ f \circ g_2 \rrparenthesis \\ \llparenthesis g_1,\ g_2 \rrparenthesis f & := & \llparenthesis g_1 \circ f,\ g_2 \circ f \rrparenthesis \\ \llparenthesis f_1,\ f_2 \rrparenthesis \bullet \llparenthesis g_1,\ g_2 \rrparenthesis & := & \llparenthesis f_1 \circ g_1,\ f_2 \circ g_2 \rrparenthesis \end{array}$$

As a reminder, this allows us to extend the idea of a function into what we'd call a *bifunction*. From there, the necessary `cpose` operators are fairly straightforward:

[22] Technically for our regular induction grammar we were using some `cposes` as well, but keeping in line with this narrative that function evaluation corresponds with endopose operators, we could readily reinterpret `distem`'s `cpose` here to be in its proper monadic form.

[23] In practice it would be more performant to implement such an environmental memory system separately.

$$\langle\, \mathit{policy?}_{\mathbf{m}},\ \mathit{next}_{\mathbf{m},1},\ \mathit{next}_{\mathbf{m},2},\ \mathit{policy?}_{\mathbf{c}},\ \mathit{next}_{\mathbf{c},1},\ \mathit{next}_{\mathbf{c},2}\,\rangle \qquad \star\{\underset{\text{call call}}{\text{open}} \,||\, \underset{\text{call call}}{\text{open}}\} \qquad (\!(\, \mathit{cont}_{\mathbf{m}},\ \mathit{cont}_{\mathbf{c}}\,)\!)$$

$$\begin{aligned} := (\!(\ \ & \text{distem}(\,\mathit{policy?}_{\mathbf{m}},\ \mathit{cont}_{\mathbf{m}},\ \mathit{next}_{\mathbf{m},1},\ -_{\mathbf{m}},\ \mathit{cont}_{\mathbf{m}},\ \mathit{next}_{\mathbf{m},2},\ -_{\mathbf{m}}\,) \quad , \\ & \text{distem}(\,\mathit{policy?}_{\mathbf{c}},\ \mathit{cont}_{\mathbf{c}},\ \mathit{next}_{\mathbf{c},1},\ -_{\mathbf{c}},\ \mathit{cont}_{\mathbf{c}},\ \mathit{next}_{\mathbf{c},2},\ -_{\mathbf{c}}\,) \qquad)\!) \end{aligned}$$

Keep in mind a quick combinatorial analysis suggests there would actually be 81 `distem` pairs given there are 9 `distem cposes`. Moreover, we expect the two columns to communicate, for example $\mathit{next}_{\mathbf{m},1}$ accepts $-_{\mathbf{m}}$ in the above, but we might want it to accept $-_{\mathbf{c}}$ as well. This means there are multiplicatively more variations as well, too many to give here! Fortunately, as the signature problem has shown us we can instead create an induction operator to do this work for us.

There is one subtlety though: If we want to be able to to shift chain compositions from one side to the other it would be best to withhold applying the `force` operators until the final composition is returned. This means we'd secretly be working with sequences of functions instead of compositions, but this changes little in the way of the theory of this essay.

Finally, with this induction grammar we can now consolidate the previously introduced paradigms: Regular induction is straightforward, we just don't use the stack side. As for context-free induction? Upon first inspection it seems more complicated since it doesn't match the vertical paradigm we developed for regular induction. Actually, it's only a matter of putting restrictions on the columns of this new grammar: In particular we allow ourselves to be able to pop from the front of the `memory` column, but we don't allow ourselves to pop or erase any functions from the `composite`.

The Nature of Non-Halting Functions

I thought I'd end this section with a mention about the *non-halting threads* which are also part of this tapestry of register functions. Such threads are now "in the mix" so to speak because of our need to accept secondary modelling grammar for our register induction operators. Since the same grammar allows us to build both computable functions and ones which don't halt, the question needs to be asked: What is it that actually makes these expressions differ?

For starters, let's we return to the version of the factorial function that was implemented with quasi-regular induction grammar:

$$n! \qquad := \qquad (1, n)\ [\ \text{car}\ \underset{\text{isZero? cdr}}{{}_\circ|_\circ}\ (\!(\,\cdot\ ,\ \text{dec cdr})\!)\ )^{n+1}$$

We observed at the time that this was one of our first examples where function construction and evaluation were no longer independent, noting that the repetition aspect of construction is now intermixed with the evaluation input. This points to a deeper pattern about recursion, and about the nature of halting: Functions which halt do so in part because of the input given to them, but also because of *how their construction is intertwined with their evaluation.*

Such thinking potentially provides a best practice approach in determining whether a function halts or not: We start by considering a hypothetical register function which builds new parts of itself for at least some of the steps of its evaluation. Next, we would inventory those *runtime* constructed parts for which we would interpret as machine *states*—accepting that such states might now be infinite. We would then use this information to determine conditions that could create evaluative loops, and in particular unending loops.[24]

Actually, since we're talking best practices the better approach would be to first use the regular and context-free pumping lemmas to rule out a given register function as being known to halt. Then we might try the just discussed strategy. With that said, it should be acknowledged with humility here that the halting problem is not an easy one to mitigate in general.

The easiest example to demonstrate the complexities of halting is the **Collatz conjecture**, where we define the function:

$$\text{Collatz}(n) \quad := \quad \begin{cases} \frac{n}{2} & \text{if } n \text{ is even,} \\ 3n+1 & \text{otherwise.} \end{cases}$$

[24]This is not to say we have a universal algorithm here—the halting problem negates that possibility—rather it is only meant to support a broader range of strategies which could be used to mitigate the more complex cases of the halting problem, ones that were previously inaccessible to existing analytic approaches.

The conjecture says that for all $n \in \mathbb{N}$ this function always reaches the value 1. If we want this to specifically be a conjecture about halting, we would first observe that by starting with value $n = 1$ and computing the next few values we end up with the following sequence:

$$1, 4, 2, 1$$

This function clearly doesn't halt, but it does repeat ad infinitum in a predictable way, and so we could just as well modify a version of it that recursively breaks at the value 1 for every input. The underlying semantics of the conjecture would not change. Either way—and this is the humbling point being made here—this conjecture has been known since at least 1937 and in all that time no one has proven or disproven it.

Afterthoughts

We've reached the end, and I'd like to share two more thematic ideas as my way of concluding this essay.

First, there was a lot to take in, in terms of this document being 52 pages. I'd like to summarize the major concepts presented here, but instead of just rehashing what was already said let's interpret it from the perspective of *skill development*, and the stages one goes through when becoming language fluent as a programmer and designer:

1. When we learn programming initially, we learn grammar to build functions to manipulate objects, and that's good.
2. Next, we often take things to a "higher order" level with functional programming, where the functions we build can now manipulate other functions, and that's good too.
3. From there, we learn things like type theory which help us to get better at function design, as they guide us in making sure our functions are well formed—of course any experienced coder can tell you error messages and debuggers teach us these things as well.
4. Beyond that, we take abstraction to a new level with things like category theory, where we learn how to recognize and prove grammars which can be used to systematically build further grammars. For example if we have some initial grammar with some properties, we might have a theorem (monadic constructions come to mind) that tell us we (or our compilers) can then build other well-behaved functions or grammars.

This sequence of developmental stages is missing a key component: If we look at all possible computable functions we can build, even having theorems about how grammatical constructs relate and can be built from each other, we might still want to know how such a space of computable functions can be organized, characterized, categorized, classified. Current pedagogies don't always make this clear.

With that said, this narrative staging is not intended to put this theory of function induction as the peak of programming. In fact it is most naturally an extension of what's known as **concatenative programming** which in many ways is considered a low level approach (before stage two) to building higher order functions.

This brings me to my second thematic idea: Most people don't code in assembly, and in practice people might not want to code in what is otherwise low level register induction grammar either. In that case, these lower level grammars provide important theoretical foundations, but in the long run we might want to return to higher level paradigms such as functional programming which have more user-friendly interfaces.

Then again, we should take a step back and consider what that might mean: It would mean we are at a point of abstraction where we are building induction grammars that build induction grammars! The absurd thing about it is that such a consideration isn't even absurd any more. The conclusion here must be that none of this essay is the end of things, and all of it is only the beginning.

Finally, I would like to acknowledge the theories (with their plethora of concepts, definitions, theorems, and proofs) for which this essay could not exist without: Type Theory, Category Theory, Automata Theory, Lambda Calculus. These endeavours represent the hard work of many very smart people over many years. Thank you.

Pijariiqpunga.

Appendix: CPose Optimizations

Stem CPose

$$\langle\, \mathit{true?},\ \mathit{break},\ \mathit{next} \,\rangle \quad \underset{\texttt{call call}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{stem}(\, \mathit{true?},\ \mathit{break},\ -_{arg},\ \mathit{cont},\ \mathit{next},\ -_{arg} \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{break} \,\rangle \quad \underset{\texttt{call id}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{dihold}(\, \mathit{true?},\ \mathit{break},\ -_{arg},\ \mathit{cont},\ -_{arg} \,)$$

$$\langle\, \mathit{true?},\ \mathit{next} \,\rangle \quad \underset{\texttt{id call}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{pend}(\, \mathit{true?},\ -_{arg},\ \mathit{cont},\ \mathit{next},\ -_{arg} \,)$$

$$\langle\, \mathit{true?},\ \mathit{break},\ x \,\rangle \quad \underset{\texttt{call pass}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{stem}(\, \mathit{true?},\ \mathit{break},\ -_{arg},\ \mathit{cont},\ -_{arg},\ x \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{break} \,\rangle \quad \underset{\texttt{call id}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{dihold}(\, \mathit{true?},\ \mathit{break},\ -_{arg},\ \mathit{cont},\ -_{arg} \,)$$

$$\langle\, \mathit{true?},\ x \,\rangle \quad \underset{\texttt{id pass}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{pend}(\, \mathit{true?},\ -_{arg},\ \mathit{cont},\ -_{arg},\ x \,)$$

$$\langle\, \mathit{true?},\ \mathit{break},\ \mathit{next},\ x \,\rangle \quad \underset{\texttt{call pose}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{stem}(\, \mathit{true?},\ \mathit{break},\ -_{w},\ \mathit{cont},\ \mathit{next},\ x \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{break},\ \mathit{next} \,\rangle \quad \underset{\texttt{call - id}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{dihold}(\, \mathit{true?},\ \mathit{break},\ -_{w},\ \mathit{cont},\ \mathit{next} \,)$$

$$\langle\, \mathit{true?},\ \mathit{break},\ x \,\rangle \quad \underset{\texttt{call id -}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{dihold}(\, \mathit{true?},\ \mathit{break},\ -_{w},\ \mathit{cont},\ x \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ x \,\rangle \quad \underset{\texttt{id pose}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{pend}(\, \mathit{true?},\ -_{w},\ \mathit{cont},\ \mathit{next},\ x \,)$$

$$\langle\, \mathit{true?},\ w,\ \mathit{next} \,\rangle \quad \underset{\texttt{pass call}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{stem}(\, \mathit{true?},\ -_{arg},\ w,\ \mathit{cont},\ \mathit{next},\ -_{arg} \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ w \,\rangle \quad \underset{\texttt{pass id}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{dihold}(\, \mathit{true?},\ -_{arg},\ w,\ \mathit{cont},\ -_{arg} \,)$$

$$\langle\, \mathit{true?},\ \mathit{next} \,\rangle \quad \underset{\texttt{id call}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{pend}(\, \mathit{true?},\ -_{arg},\ \mathit{cont},\ \mathit{next},\ -_{arg} \,)$$

$$\langle\, \mathit{true?},\ w,\ x \,\rangle \quad \underset{\texttt{pass pass}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{stem}(\, \mathit{true?},\ -_{arg},\ w,\ \mathit{cont},\ -_{arg},\ x \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ w \,\rangle \quad \underset{\texttt{pass id}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{dihold}(\, \mathit{true?},\ -_{arg},\ w,\ \mathit{cont},\ -_{arg} \,)$$

$$\langle\, \mathit{true?},\ x \,\rangle \quad \underset{\texttt{id pass}}{\star\{\text{closing}\}} \quad \mathit{cont} \quad := \quad \mathrm{pend}(\, \mathit{true?},\ -_{arg},\ \mathit{cont},\ -_{arg},\ x \,)$$

$$\langle \mathit{true?},\ w,\ \mathit{next},\ x \rangle \quad \underset{\text{pass pose}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{stem}(\mathit{true?},\ -_{\mathit{break}},\ w,\ \mathit{cont},\ \mathit{next},\ x)$$

$$\Downarrow$$

$$\langle \mathit{true?},\ w,\ \mathit{next} \rangle \quad \underset{\text{pass } - \text{ id}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{dihold}(\mathit{true?},\ -_{\mathit{break}},\ w,\ \mathit{cont},\ \mathit{next})$$

$$\langle \mathit{true?},\ w,\ x \rangle \quad \underset{\text{pass id } -}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{dihold}(\mathit{true?},\ -_{\mathit{break}},\ w,\ \mathit{cont},\ x)$$

$$\langle \mathit{true?},\ \mathit{next},\ x \rangle \quad \underset{\text{id pose}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{pend}(\mathit{true?},\ -_{\mathit{break}},\ \mathit{cont},\ \mathit{next},\ x)$$

$$\langle \mathit{true?},\ \mathit{break},\ w,\ \mathit{next} \rangle \quad \underset{\text{pose call}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{stem}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ \mathit{next},\ -_{x})$$

$$\Downarrow$$

$$\langle \mathit{true?},\ \mathit{break},\ w \rangle \quad \underset{\text{pose id}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{dihold}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ -_{x})$$

$$\langle \mathit{true?},\ \mathit{break},\ \mathit{next} \rangle \quad \underset{- \text{ id call}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{pend}(\mathit{true?},\ \mathit{break},\ \mathit{cont},\ \mathit{next},\ -_{x})$$

$$\langle \mathit{true?},\ w,\ \mathit{next} \rangle \quad \underset{\text{id } - \text{ call}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{pend}(\mathit{true?},\ w,\ \mathit{cont},\ \mathit{next},\ -_{x})$$

$$\langle \mathit{true?},\ \mathit{break},\ w,\ x \rangle \quad \underset{\text{pose pass}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{stem}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ -_{\mathit{next}},\ x)$$

$$\Downarrow$$

$$\langle \mathit{true?},\ \mathit{break},\ w \rangle \quad \underset{\text{pose id}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{dihold}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ -_{\mathit{next}})$$

$$\langle \mathit{true?},\ \mathit{break},\ x \rangle \quad \underset{- \text{ id pass}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{pend}(\mathit{true?},\ \mathit{break},\ \mathit{cont},\ -_{\mathit{next}},\ x)$$

$$\langle \mathit{true?},\ w,\ x \rangle \quad \underset{\text{id } - \text{ pass}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{pend}(\mathit{true?},\ w,\ \mathit{cont},\ -_{\mathit{next}},\ x)$$

$$\langle \mathit{true?},\ \mathit{break},\ w,\ \mathit{next},\ x \rangle \quad \underset{\text{pose pose}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{stem}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ \mathit{next},\ x)$$

$$\Downarrow$$

$$\langle \mathit{true?},\ \mathit{break},\ w,\ \mathit{next} \rangle \quad \underset{\text{pose } - \text{ id}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{dihold}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ \mathit{next})$$

$$\langle \mathit{true?},\ \mathit{break},\ w,\ x \rangle \quad \underset{\text{pose id } -}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{dihold}(\mathit{true?},\ \mathit{break},\ w,\ \mathit{cont},\ x)$$

$$\langle \mathit{true?},\ \mathit{break},\ \mathit{next},\ x \rangle \quad \underset{- \text{ id pose}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{pend}(\mathit{true?},\ \mathit{break},\ \mathit{cont},\ \mathit{next},\ x)$$

$$\langle \mathit{true?},\ w,\ \mathit{next},\ x \rangle \quad \underset{\text{id } - \text{ pose}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{pend}(\mathit{true?},\ w,\ \mathit{cont},\ \mathit{next},\ x)$$

$$\langle \mathit{true?},\ \mathit{break},\ \mathit{next} \rangle \quad \underset{- \text{ id } - \text{ id}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{hold}(\mathit{true?},\ \mathit{break},\ \mathit{cont},\ \mathit{next})$$

$$\langle \mathit{true?},\ \mathit{break},\ x \rangle \quad \underset{- \text{ id id } -}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{hold}(\mathit{true?},\ \mathit{break},\ \mathit{cont},\ x)$$

$$\langle \mathit{true?},\ w,\ \mathit{next} \rangle \quad \underset{\text{id } - - \text{ id}}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{hold}(\mathit{true?},\ w,\ \mathit{cont},\ \mathit{next})$$

$$\langle \mathit{true?},\ w,\ x \rangle \quad \underset{\text{id } - \text{ id } -}{\star\{\text{closing}\}}\ \mathit{cont} \quad := \quad \text{hold}(\mathit{true?},\ w,\ \mathit{cont},\ x)$$

Costem CPose

$$\langle\, \mathit{true?},\ \mathit{next},\ \mathit{break}\,\rangle \quad \underset{\text{call call}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next},\ -_{arg},\ \mathit{break},\ -_{arg}\,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{next}\,\rangle \quad \underset{\text{call id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next},\ -_{arg},\ -_{arg}\,)$$

$$\langle\, \mathit{true?},\ \mathit{break}\,\rangle \quad \underset{\text{id call}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ \mathit{break},\ -_{arg}\,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ w\,\rangle \quad \underset{\text{call pass}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next},\ -_{arg},\ -_{arg},\ w\,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{next}\,\rangle \quad \underset{\text{call id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next},\ -_{arg},\ -_{arg}\,)$$

$$\langle\, \mathit{true?},\ w\,\rangle \quad \underset{\text{id pass}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ -_{arg},\ w\,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ \mathit{break},\ w\,\rangle \quad \underset{\text{call pose}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next},\ -_{x},\ \mathit{break},\ w\,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{next},\ \mathit{break}\,\rangle \quad \underset{\text{call}\ -\ \text{id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next},\ -_{x},\ \mathit{break}\,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ w\,\rangle \quad \underset{\text{call id}\ -}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\,\mathit{true?},\ \mathit{cont},\ \mathit{next},\ -_{x},\ w\,)$$

$$\langle\, \mathit{true?},\ \mathit{break},\ w\,\rangle \quad \underset{\text{id pose}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\,\mathit{true?},\ \mathit{cont},\ -_{x},\ \mathit{break},\ w\,)$$

$$\langle\, \mathit{true?},\ x,\ \mathit{break}\,\rangle \quad \underset{\text{pass call}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ x,\ \mathit{break},\ -_{arg}\,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ x\,\rangle \quad \underset{\text{pass id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ x,\ -_{arg}\,)$$

$$\langle\, \mathit{true?},\ \mathit{break}\,\rangle \quad \underset{\text{id call}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ \mathit{break},\ -_{arg}\,)$$

$$\langle\, \mathit{true?},\ x,\ w\,\rangle \quad \underset{\text{pass pass}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ x,\ -_{arg},\ w\,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ x\,\rangle \quad \underset{\text{pass id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ x,\ -_{arg}\,)$$

$$\langle\, \mathit{true?},\ w\,\rangle \quad \underset{\text{id pass}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\,\mathit{true?},\ \mathit{cont},\ -_{arg},\ -_{arg},\ w\,)$$

$$\langle\, \mathit{true?},\ x,\ \mathit{break},\ w \,\rangle \quad \underset{\texttt{pass pose}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\, \mathit{true?},\ \mathit{cont},\ -_{\mathit{next}},\ x,\ \mathit{break},\ w \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ x,\ \mathit{break} \,\rangle \quad \underset{\texttt{pass - id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\, \mathit{true?},\ \mathit{cont},\ -_{\mathit{next}},\ x,\ \mathit{break} \,)$$

$$\langle\, \mathit{true?},\ x,\ w \,\rangle \quad \underset{\texttt{pass id -}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\, \mathit{true?},\ \mathit{cont},\ -_{\mathit{next}},\ x,\ w \,)$$

$$\langle\, \mathit{true?},\ \mathit{break},\ w \,\rangle \quad \underset{\texttt{id pose}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\, \mathit{true?},\ \mathit{cont},\ -_{\mathit{next}},\ \mathit{break},\ w \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ x,\ \mathit{break} \,\rangle \quad \underset{\texttt{pose call}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ x,\ \mathit{break},\ -_{w} \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{next},\ x \,\rangle \quad \underset{\texttt{pose id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ x,\ -_{w} \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ \mathit{break} \,\rangle \quad \underset{\texttt{- id call}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ \mathit{break},\ -_{w} \,)$$

$$\langle\, \mathit{true?},\ x,\ \mathit{break} \,\rangle \quad \underset{\texttt{id - call}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\, \mathit{true?},\ \mathit{cont},\ x,\ \mathit{break},\ -_{w} \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ x,\ w \,\rangle \quad \underset{\texttt{pose pass}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ x,\ -_{\mathit{break}},\ w \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{next},\ x \,\rangle \quad \underset{\texttt{pose id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ x,\ -_{\mathit{break}} \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ w \,\rangle \quad \underset{\texttt{- id pass}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ -_{\mathit{break}},\ w \,)$$

$$\langle\, \mathit{true?},\ x,\ w \,\rangle \quad \underset{\texttt{id - pass}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\, \mathit{true?},\ \mathit{cont},\ x,\ -_{\mathit{break}},\ w \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ x,\ \mathit{break},\ w \,\rangle \quad \underset{\texttt{pose pose}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{costem}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ x,\ \mathit{break},\ w \,)$$

$$\Downarrow$$

$$\langle\, \mathit{true?},\ \mathit{next},\ x,\ \mathit{break} \,\rangle \quad \underset{\texttt{pose - id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ x,\ \mathit{break} \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ x,\ w \,\rangle \quad \underset{\texttt{pose id -}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{copend}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ x,\ w \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ \mathit{break},\ w \,\rangle \quad \underset{\texttt{- id pose}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ \mathit{break},\ w \,)$$

$$\langle\, \mathit{true?},\ x,\ \mathit{break},\ w \,\rangle \quad \underset{\texttt{id - pose}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{dihold}(\, \mathit{true?},\ \mathit{cont},\ x,\ \mathit{break},\ w \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ \mathit{break} \,\rangle \quad \underset{\texttt{- id - id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{cohold}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ \mathit{break} \,)$$

$$\langle\, \mathit{true?},\ \mathit{next},\ w \,\rangle \quad \underset{\texttt{- id id -}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{cohold}(\, \mathit{true?},\ \mathit{cont},\ \mathit{next},\ w \,)$$

$$\langle\, \mathit{true?},\ x,\ \mathit{break} \,\rangle \quad \underset{\texttt{id - - id}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{cohold}(\, \mathit{true?},\ \mathit{cont},\ x,\ \mathit{break} \,)$$

$$\langle\, \mathit{true?},\ x,\ w \,\rangle \quad \underset{\texttt{id - id -}}{\star\{\text{opening}\}} \quad \mathit{cont} \quad := \quad \text{cohold}(\, \mathit{true?},\ \mathit{cont},\ x,\ w \,)$$

Distem CPose

$$\begin{array}{llcl}
\langle\, true?,\ next_1,\ next_2 \,\rangle & \underset{\text{call call}}{\star\{\text{open}\}}\ cont & := & \text{distem}(\, true?,\ cont,\ next_1,\ -_{arg},\ cont,\ next_2,\ -_{arg} \,) \\
 & & \Downarrow & \\
\langle\, true?,\ next_1 \,\rangle & \underset{\text{call id}}{\star\{\text{open}\}}\ cont & = & \text{costem}(\, true?,\ cont,\ next_1,\ -_{arg},\ cont,\ -_{arg} \,) \\
\langle\, true?,\ next_2 \,\rangle & \underset{\text{id call}}{\star\{\text{open}\}}\ cont & = & \text{stem}(\, true?,\ cont,\ -_{arg},\ cont,\ next_2,\ -_{arg} \,)
\end{array}$$

$$\begin{array}{llcl}
\langle\, true?,\ next_1,\ x_2 \,\rangle & \underset{\text{call pass}}{\star\{\text{open}\}}\ cont & := & \text{distem}(\, true?,\ cont,\ next_1,\ -_{arg},\ cont,\ -_{arg},\ x_2 \,) \\
 & & \Downarrow & \\
\langle\, true?,\ next_1 \,\rangle & \underset{\text{call id}}{\star\{\text{open}\}}\ cont & = & \text{costem}(\, true?,\ cont,\ next_1,\ -_{arg},\ cont,\ -_{arg} \,) \\
\langle\, true?,\ x_2 \,\rangle & \underset{\text{id pass}}{\star\{\text{open}\}}\ cont & = & \text{stem}(\, true?,\ cont,\ -_{arg},\ cont,\ -_{arg},\ x_2 \,)
\end{array}$$

$$\begin{array}{llcl}
\langle\, true?,\ next_1,\ next_2,\ x_2 \,\rangle & \underset{\text{call pose}}{\star\{\text{open}\}}\ cont & := & \text{distem}(\, true?,\ cont,\ next_1,\ -_{x_1},\ cont,\ next_2,\ x_2 \,) \\
 & & \Downarrow & \\
\langle\, true?,\ next_1,\ next_2 \,\rangle & \underset{\text{call } - \text{ id}}{\star\{\text{open}\}}\ cont & = & \text{costem}(\, true?,\ cont,\ next_1,\ -_{x_1},\ cont,\ next_2 \,) \\
\langle\, true?,\ next_1,\ x_2 \,\rangle & \underset{\text{call id } -}{\star\{\text{open}\}}\ cont & = & \text{costem}(\, true?,\ cont,\ next_1,\ -_{x_1},\ cont,\ x_2 \,) \\
\langle\, true?,\ next_2,\ x_2 \,\rangle & \underset{\text{id pose}}{\star\{\text{open}\}}\ cont & = & \text{stem}(\, true?,\ cont,\ -_{x_1},\ cont,\ next_2,\ x_2 \,)
\end{array}$$

$$\begin{array}{llcl}
\langle\, true?,\ x_1,\ next_2 \,\rangle & \underset{\text{pass call}}{\star\{\text{open}\}}\ cont & := & \text{distem}(\, true?,\ cont,\ -_{arg},\ x_1,\ cont,\ next_2,\ -_{arg} \,) \\
 & & \Downarrow & \\
\langle\, true?,\ x_1 \,\rangle & \underset{\text{pass id}}{\star\{\text{open}\}}\ cont & = & \text{costem}(\, true?,\ cont,\ -_{arg},\ x_1,\ cont,\ -_{arg} \,) \\
\langle\, true?,\ next_2 \,\rangle & \underset{\text{id call}}{\star\{\text{open}\}}\ cont & = & \text{stem}(\, true?,\ cont,\ -_{arg},\ cont,\ next_2,\ -_{arg} \,)
\end{array}$$

$$\begin{array}{llcl}
\langle\, true?,\ x_1,\ x_2 \,\rangle & \underset{\text{pass pass}}{\star\{\text{open}\}}\ cont & := & \text{distem}(\, true?,\ cont,\ -_{arg},\ x_1,\ cont,\ -_{arg},\ x_2 \,) \\
 & & \Downarrow & \\
\langle\, true?,\ x_1 \,\rangle & \underset{\text{pass id}}{\star\{\text{open}\}}\ cont & = & \text{costem}(\, true?,\ cont,\ -_{arg},\ x_1,\ cont,\ -_{arg} \,) \\
\langle\, true?,\ x_2 \,\rangle & \underset{\text{id pass}}{\star\{\text{open}\}}\ cont & = & \text{stem}(\, true?,\ cont,\ -_{arg},\ cont,\ -_{arg},\ x_2 \,)
\end{array}$$

$$\langle \mathit{true?},\ x_1,\ \mathit{next}_2,\ x_2 \rangle \quad \underset{\text{pass pose}}{\star\{\text{open}\}}\ \mathit{cont} \quad := \quad \text{distem}(\mathit{true?},\ \mathit{cont},\ -_{\mathit{next}_1},\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ x_2)$$

$$\Downarrow$$

$$\langle \mathit{true?},\ x_1,\ \mathit{next}_2 \rangle \quad \underset{\text{pass} - \text{id}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{costem}(\mathit{true?},\ \mathit{cont},\ -_{\mathit{next}_1},\ x_1,\ \mathit{cont},\ \mathit{next}_2)$$

$$\langle \mathit{true?},\ x_1,\ x_2 \rangle \quad \underset{\text{pass id} -}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{costem}(\mathit{true?},\ \mathit{cont},\ -_{\mathit{next}_1},\ x_1,\ \mathit{cont},\ x_2)$$

$$\langle \mathit{true?},\ \mathit{next}_2,\ x_2 \rangle \quad \underset{\text{id pose}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{stem}(\mathit{true?},\ \mathit{cont},\ -_{x_1},\ \mathit{cont},\ \mathit{next}_2,\ x_2)$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_1,\ \mathit{next}_2 \rangle \quad \underset{\text{pose call}}{\star\{\text{open}\}}\ \mathit{cont} \quad := \quad \text{distem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ -_{x_2})$$

$$\Downarrow$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_1 \rangle \quad \underset{\text{pose id}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{costem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ -_{x_2})$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ \mathit{next}_2 \rangle \quad \underset{- \text{ id call}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{stem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ \mathit{cont},\ \mathit{next}_2,\ -_{x_2})$$

$$\langle \mathit{true?},\ x_1,\ \mathit{next}_2 \rangle \quad \underset{\text{id} - \text{call}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{stem}(\mathit{true?},\ \mathit{cont},\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ -_{x_2})$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_1,\ x_2 \rangle \quad \underset{\text{pose pass}}{\star\{\text{open}\}}\ \mathit{cont} \quad := \quad \text{distem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ -_{\mathit{next}_2},\ x_2)$$

$$\Downarrow$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_1 \rangle \quad \underset{\text{pose id}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{costem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ -_{\mathit{next}_2})$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_2 \rangle \quad \underset{- \text{ id pass}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{stem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ \mathit{cont},\ -_{\mathit{next}_2},\ x_2)$$

$$\langle \mathit{true?},\ x_1,\ x_2 \rangle \quad \underset{\text{id} - \text{pass}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{stem}(\mathit{true?},\ \mathit{cont},\ x_1,\ \mathit{cont},\ -_{\mathit{next}_2},\ x_2)$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_1,\ \mathit{next}_2,\ x_2 \rangle \quad \underset{\text{pose pose}}{\star\{\text{open}\}}\ \mathit{cont} \quad := \quad \text{distem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ x_2)$$

$$\Downarrow$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_1,\ \mathit{next}_2 \rangle \quad \underset{\text{pose} - \text{id}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{costem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ \mathit{next}_2)$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_1,\ x_2 \rangle \quad \underset{\text{pose id} -}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{costem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ x_1,\ \mathit{cont},\ x_2)$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ \mathit{next}_2,\ x_2 \rangle \quad \underset{- \text{ id pose}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{stem}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ \mathit{cont},\ \mathit{next}_2,\ x_2)$$

$$\langle \mathit{true?},\ x_1,\ \mathit{next}_2,\ x_2 \rangle \quad \underset{\text{id} - \text{pose}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{stem}(\mathit{true?},\ \mathit{cont},\ x_1,\ \mathit{cont},\ \mathit{next}_2,\ x_2)$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ \mathit{next}_2 \rangle \quad \underset{- \text{ id} - \text{id}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{dihold}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ \mathit{cont},\ \mathit{next}_2)$$

$$\langle \mathit{true?},\ \mathit{next}_1,\ x_2 \rangle \quad \underset{- \text{ id id} -}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{dihold}(\mathit{true?},\ \mathit{cont},\ \mathit{next}_1,\ \mathit{cont},\ x_2)$$

$$\langle \mathit{true?},\ x_1,\ \mathit{next}_2 \rangle \quad \underset{\text{id} - - \text{id}}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{dihold}(\mathit{true?},\ \mathit{cont},\ x_1,\ \mathit{cont},\ \mathit{next}_2)$$

$$\langle \mathit{true?},\ x_1,\ x_2 \rangle \quad \underset{\text{id} - \text{id} -}{\star\{\text{open}\}}\ \mathit{cont} \quad = \quad \text{dihold}(\mathit{true?},\ \mathit{cont},\ x_1,\ \mathit{cont},\ x_2)$$

References

[1] Homotopy Type Theory: Univalent Foundations of Mathematics. The Univalent Foundations Program (2013).

[2] D. Nikpayuk. Toward the Semantic Reconstruction of Mathematical Functions (2020). https://github.com/Daniel-Nikpayuk/Mathematics/blob/main/Essays/Function%20Semantics/Version-Two/semantics.pdf

[3] J.E. Hopcroft, R. Motwani, J.D. Ullman. Introduction to Automata Theory, Languages, and Computation (second edition). Addison-Wesley Publishing (2001).

[4] H. Abelson, G.J. Sussman. Structure and Interpretation of Computer Programs (second edition). The Massachusetts Institute of Technology Press (1996).

[5] E. Riehl. Category Theory in Context. Dover Publications, Inc. (2016).