

# Tidyparse: Real-Time Context-free Error Correction

**Breandan Considine**, Jin Guo, Xujie Si

McGill University, Mila IQIA

*bre@ndan.co*

November 23, 2022

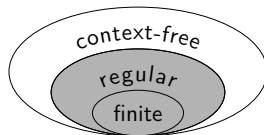


# Overview

- 1 Algebraic Parsing
- 2 Typelevel Programming
- 3 Graph Programming
- 4 Random Numbers
- 5 Finite Fields
- 6 Future Work

# Background: Regular grammars

A regular grammar is a 4-tuple  $G = \langle V, \Sigma, P, S \rangle$  where  $V$  is the set of nonterminals,  $\Sigma$  are the terminals,  $P : V \times (V \cup \Sigma)^{\leq 2}$  are the productions, and  $S \in V$  is the start symbol, i.e., all productions are of the form  $A \rightarrow a$ ,  $A \rightarrow aB$  (right-regular), or  $A \rightarrow Ba$  (left-regular). Regular languages are closed under union, complementation, concatenation, and Kleene star.

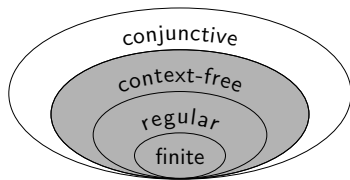


# Background: Context-free grammars

In a context-free grammar  $G = \langle V, \Sigma, P, S \rangle$  all productions are of the form  $P : V \times (V \cup \Sigma)^+$ , i.e., RHS may contain any number of nonterminals,  $V$ .

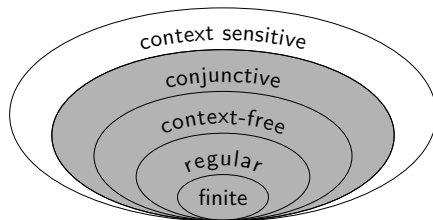
For example, consider the grammar  $S \rightarrow SS \mid (S) \mid ()$ . This represents the language of balanced parentheses, e.g.  $()$ ,  $()()$ ,  $((()))$ ,  $()((()))$ ,  $((()()))$ ,  $((()))()$ ...

Every CFG has a normal form  $P^* : V \times (V^2 \mid \Sigma)$ , i.e., every production can be refactored into either  $v_0 \rightarrow v_1 v_2$  or  $v_0 \rightarrow \sigma$ , where  $v_0 \dots v_2 : V$  and  $\sigma : \Sigma$ , e.g.,  $\{S \rightarrow SS \mid (S) \mid ()\} \Leftrightarrow^* \{S \rightarrow XR \mid SS \mid LR, L \rightarrow (, R \rightarrow ), X \rightarrow LS\}$



# Background: Conjunctive grammars

Conjunctive grammars naturally extend CFGs with language union and intersection, respecting closure under those operations. Equivalent to trellis automata, which are like contractive elementary cellular automata. Language inclusion is decidable in P.



# Background: Closure properties of formal languages

Formal languages are not closed, e.g.,  $\text{CFL} \cap \text{CFL}$  is not CFL in general.  
Let  $\cdot$  denote concatenation,  $\star$  be Kleene star, and  $\complement$  be complementation:

	$\cup$	$\cap$	$\cdot$	$\star$	$\complement$
Finite	✓	✓	✓	✓	✓
Regular	✓	✓	✓	✓	✓
Context-free	✓	✗	✓	✓	✗
Conjunctive	✓	✓	✓	✓	?
Context-sensitive	✓	✓	✓	+	✓
Recursively Enumerable	✓	✓	✓	✓	✗

# Context-free parsing, distilled

Given a CFG  $\mathcal{G} := \langle V, \Sigma, P, S \rangle$  in Chomsky Normal Form, we can construct a recognizer  $R_{\mathcal{G}} : \Sigma^n \rightarrow \mathbb{B}$  for strings  $\sigma : \Sigma^n$  as follows. Let  $2^V$  be our domain,  $0$  be  $\emptyset$ ,  $\oplus$  be  $\cup$ , and  $\otimes$  be defined as follows:

$$s_1 \otimes s_2 := \{ C \mid \langle A, B \rangle \in s_1 \times s_2, (C \rightarrow AB) \in P \}$$

e.g.,  $\{A \rightarrow BC, C \rightarrow AD, D \rightarrow BA\} \subseteq P \vdash \{A, B, C\} \otimes \{B, C, D\} = \{A, C\}$

If we define  $\sigma_r^{\rightarrow} := \{w \mid (w \rightarrow \sigma_r) \in P\}$ , then initialize

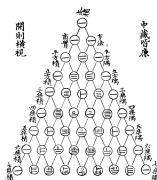
$M_{r+1=c}^0(\mathcal{G}', e) := \sigma_r^{\rightarrow}$  and solve for the fixpoint  $M^* = M + M^2$ ,

$$M^0 := \begin{pmatrix} \emptyset & \sigma_1^{\rightarrow} & \emptyset & \cdots & \emptyset \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \emptyset & \cdots & \cdots & \cdots & \sigma_n^{\rightarrow} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \emptyset & \cdots & \cdots & \cdots & \emptyset \end{pmatrix} \Rightarrow M^* = \begin{pmatrix} \emptyset & \sigma_1^{\rightarrow} & \Lambda & \cdots & \Lambda_{\sigma}^* \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \emptyset & \cdots & \cdots & \cdots & \sigma_n^{\rightarrow} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \emptyset & \cdots & \cdots & \cdots & \emptyset \end{pmatrix}$$

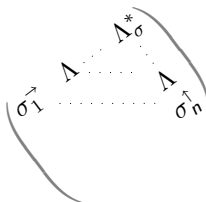
Valiant (1975) shows that  $\sigma \in \mathcal{L}(\mathcal{G})$  iff  $S \in \mathcal{T}$ , i.e.,  $\mathbb{1}_{\mathcal{T}}(S) \iff \mathbb{1}_{\mathcal{L}(\mathcal{G})}(\sigma)$ .

# Lattices, Matrices and Trellises

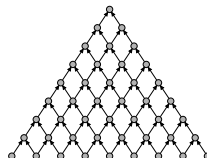
The art of treillage has been practiced from ancient through modern times.



Jia Xian Triangle  
Jia, ~1030 A.D.



CYK Parsing  
Sakai, 1961 A.D.



Trellis Automaton  
Dyer, 1980 A.D.



# A few observations on algebraic parsing

- The matrix  $\mathbf{M}^*$  is strictly upper triangular, i.e., nilpotent of degree  $n$
- Recognizer can be translated into a parser by storing backpointers

$$\mathbf{M}_1 = \mathbf{M}_0 + \mathbf{M}_0^2$$


$$\mathbf{M}_2 = \mathbf{M}_1 + \mathbf{M}_1^2$$


$$\mathbf{M}_3 = \mathbf{M}_2 + \mathbf{M}_2^2 = \mathbf{M}_4$$


- The  $\otimes$  operator is *not* associative:  $S \otimes (S \otimes S) \neq (S \otimes S) \otimes S$
- Built-in error recovery: nonempty submatrices = parsable fragments
- `seekFixpoint { it + it * it }` is sufficient but unnecessary
- If we had a way to solve for  $\mathbf{M} = \mathbf{M} + \mathbf{M}^2$  directly, power iteration would be unnecessary, could solve for  $\mathbf{M} = \mathbf{M}^2$  above superdiagonal

# Satisfiability + holes (our contribution)

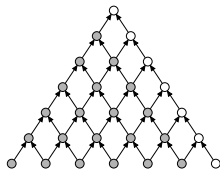
- Can be lowered onto a Boolean tensor  $\mathbb{B}_2^{n \times n \times |V|}$  (Valiant, 1975)
- Binarized CYK parser can be efficiently compiled to a SAT solver
- Enables sketch-based synthesis in either  $\sigma$  or  $\mathcal{G}$ : just use variables!
- We simply encode the characteristic function, i.e.  $\mathbb{1}_{\subseteq V} : V \rightarrow \mathbb{Z}_2^{|V|}$
- $\oplus, \otimes$  are defined as  $\boxplus, \boxtimes$ , so that the following diagram commutes:

$$\begin{array}{ccc} 2^V \times 2^V & \xrightarrow{\oplus/\otimes} & 2^V \\ \mathbb{1}^{-2} \uparrow \mathbb{1}^2 & & \mathbb{1}^{-1} \uparrow \mathbb{1} \\ \mathbb{Z}_2^{|V|} \times \mathbb{Z}_2^{|V|} & \xrightarrow{\boxplus/\boxtimes} & \mathbb{Z}_2^{|V|} \end{array}$$

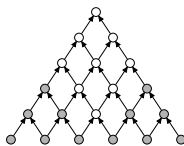
- These operators can be lifted into matrices/tensors in the usual way
- In most cases, only a few nonterminals are active at any given time
- More sophisticated representations are known for  $\binom{n}{0 \leq k}$  subsets
- If density is desired, possible to use the Maculay representation
- If you know of a more efficient encoding, please let us know!

# Incremental parsing

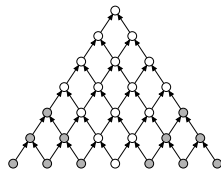
Should only need to recompute submatrices affected by individual edits. In the worst case, each edit requires quadratic complexity in terms of  $|\Sigma^*|$ .



Append  
 $\mathcal{O}(n)$



Delete  
 $\mathcal{O}\left(\frac{1}{4}(n-1)^2\right)$



Insert  
 $\mathcal{O}\left(\frac{1}{4}(n+1)^2\right)$

Related to **dynamic matrix inverse** and **incremental transitive closure** with vertex updates. With a careful encoding, we can incrementally update SAT constraints as new keystrokes are received to eliminate redundancy.

# Conjunctive parsing

It is well-known that the family of CFLs is not closed under intersection. For example, consider  $\mathcal{L}_\cap := \mathcal{L}(\mathcal{G}_1) \cap \mathcal{L}(\mathcal{G}_2)$ :

$$P_1 := \{ S \rightarrow LR, \quad L \rightarrow ab \mid aLb, \quad R \rightarrow c \mid cR \}$$

$$P_2 := \{ S \rightarrow LR, \quad R \rightarrow bc \mid bRc, \quad L \rightarrow a \mid aL \}$$

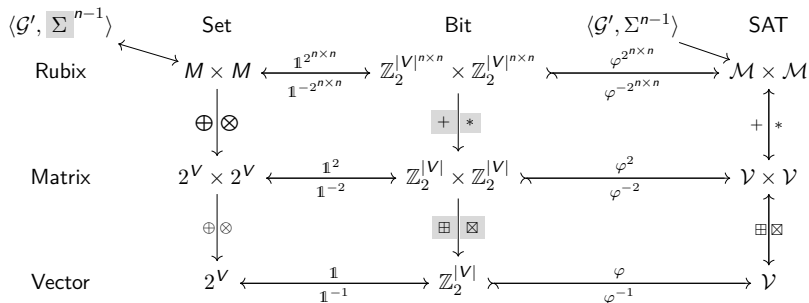
Note that  $\mathcal{L}_\cap$  generates the language  $\{ a^d b^d c^d \mid d > 0 \}$ , which according to the pumping lemma is not context-free. To encode  $\mathcal{L}_\cap$ , we intersect all terminals  $\Sigma_\cap := \bigcap_{i=1}^c \Sigma_i$ , then for each  $t_\cap \in \Sigma_\cap$  and CFG, construct an equivalence class  $E(t_\cap, \mathcal{G}_i) = \{ w_i \mid (w_i \rightarrow t_\cap) \in P_i \}$  as follows:

$$\bigwedge_{t \in \Sigma_\cap} \bigwedge_{j=1}^{c-1} \bigwedge_{i=1}^{|\sigma|} E(t_\cap, \mathcal{G}_j) \equiv_{\sigma_i} E(t_\cap, \mathcal{G}_{j+1}) \quad (1)$$



# A birds eye view of the algorithm

We can lower Valiant's algorithm onto a polynomial system of equations over finite fields, allowing us to solve for holes and parse trees.



So far, we only consider Cartesian closed categories, however, we can also consider other categories, such as the category of CFLs under conjunction, which allows us to encode the intersection of two CFGs.

# Kotlin implementation: CFG definition

```
typealias Production = Pair<String, List<String>>
typealias CFG = Set<Production>
val Production.LHS: String get() = first
val Production.RHS: List<String> get() = second
val CFG.nonterminals: Set<String> by cache { map { it.LHS }.toSet() }
val CFG.words: Set<String> by cache { nonterminals + flatMap { it.RHS } }
val CFG.terminals: Set<String> by cache { words - nonterminals }
// Many-to-many mapping of nonterminals to RHS expansions
val CFG.bimap: BidirectionalMap by cache { BidirectionalMap(this) }

fun CFG.makeAlgebra(): Ring<Set<String>> =
    Ring.of(
        //  $\emptyset = \emptyset$ 
        nil = setOf(),
        //  $x + y = x \cup y$ 
        plus = { x, y -> x union y },
        //  $x \cdot y = \{ A\emptyset \mid A1 \in x, A2 \in y, (A\emptyset \rightarrow A1 A2) \in P \}$ 
        times = { x, y -> join(x, y) }
    )

fun CFG.join(ls: Set<String>, rs: Set<String>): Set<String> =
    (ls * rs).flatMap { (l, r) -> bimap[listOf(l, r)] }.toSet()
```

# Kotlin implementation: the recognizer

```
// Constructs initial matrix according to:  $M_{i+1=j} = \{ A \mid (A \rightarrow \sigma_i) \in P \}$ 
fun CFG.initialMatrix(str: List<String>): Matrix<Set<String>> =
    Matrix(makeAlgebra(), str.size + 1) { i, j ->
        // Aligns nonterminals matching each terminal along superdiagonal
        if (i + 1  $\neq$  j) emptySet() else bimap(listOf(str[j - 1])).toSet()
    }

// Computes the fixpoint of an abstract matrix function
tailrec fun <T: Matrix<S>, S> T.seekFixpoint(op: (T) -> T): T {
    val next = op(this)
    return if (this == next) next else next.seekFixpoint(op)
}

// Checks whether start symbol is contained in the northeasternmost entry
fun CFG.check(s: String): Boolean = START in parse(tokenize(s))[0].last()

// Since matrix is strictly UT, this converges in at most |tokens| steps
fun CFG.parse(tokens: List<String>): Matrix<Set<String>> =
    initialMatrix(tokens).seekFixpoint { it + it * it }
```

# Tidyparse IDE plugin

The screenshot displays the Tidyparse IDE plugin interface. At the top, a list of open files includes `arithmetic_left_recurse/arithmetic.tidy`, `mini_ocaml/ocaml.tidy`, `arithmetic_checked/arithmetic.tidy`, `arithmetic.cfg`, `simple.tidy`, `simple.cfg`, `test.tidy`, `ocaml/ocaml.tidy`, and `SATValiantTest.at`. The main editor shows a code completion menu for the expression `let rec a =`. The menu lists several options: `( <X> , <X> )`, `( <X> , [] )`, `( <X> , filter )`, `( [] , [] )`, and `( [] , filter )`. Below the menu, the text `Press <?> to insert, => to replace` is visible. The bottom panel shows two views of the `ocaml.cfg` file. The left view displays grammar rules for `S`, `X`, `A`, `FUN`, `F`, `M`, `Branch`, `L`, and `LI`. The right view displays grammar rules for `V`, `Vexp`, `Vname`, `FunName`, `V0`, `I`, and `B`.

```
let rec a = _ _ _ _ _

let rec filter p l =
  let rec a = ( <X> , <X> )
  let rec a = ( <X> , [] )
  let rec a = ( <X> , filter )
  let rec a = ( [] , [] )
  let rec a = ( [] , filter )

  l -> if p x then x :: _ _ _ else

let curry f = ( fun x

S -> X
X -> A | V | ( X , X ) | X X | ( X )
A -> FUN | F | LI | M | L
FUN -> fun V `->` X
F -> if X then X else X
M -> match V with Branch
Branch -> `|` X `->` X | Branch Branch
L -> let V = X
L -> let rec V = X
LI -> L in X

V -> Vexp | ( Vexp ) | List | Vexp Vexp
Vexp -> Vname | FunName | Vexp V0 Vexp | B
Vexp -> ( Vname , Vname ) | Vexp Vexp | I
List -> [] | V :: V
Vname -> a | b | c | d | e | f | g | h | i
Vname -> j | k | l | m | n | o | p | q | r
Vname -> s | t | u | v | w | x | y | z
FunName -> foldright | map | filter
FunName -> curry | uncurry | ( V0 )
V0 -> + | - | * | / | >
V0 -> = | < | `|` | `&&`
I -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
B -> true | false
```

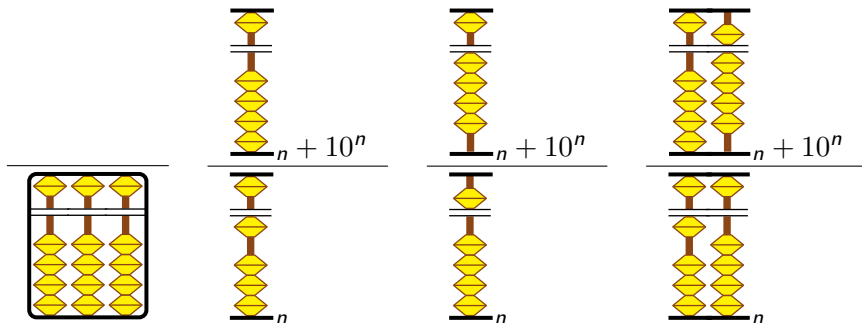


# Abbreviated history of algebraic parsing

- Chomsky & Schützenberger (1959) - The algebraic theory of CFLs
- Cocke–Younger–Kasami (1961) - Bottom-up matrix-based parsing
- Brzozowski (1964) - Derivatives of regular expressions
- Earley (1968) - top-down dynamic programming (no CNF needed)
- Valiant (1975) - first realizes the Boolean matrix correspondence
  - Naïvely, has complexity  $\mathcal{O}(n^4)$ , can be reduced to  $\mathcal{O}(n^\omega)$ ,  $\omega < 2.763$
- Lee (1997) - Fast CFG Parsing  $\iff$  Fast BMM, formalizes reduction
- Might et al. (2011) - Parsing with derivatives (Brzozowski  $\Rightarrow$  CFL)
- Bakinova, Okhotin et al. (2010) - Formal languages over GF(2)
- Bernady & Jansson (2015) - Certifies Valiant (1975) in Agda
- Cohen & Gildea (2016) - Generalizes Valiant (1975) to parse and recognize mildly context sensitive languages, e.g. LCFRS, TAG, CCG
- **Considine, Guo & Si (2022) - SAT + Valiant (1975) + holes**

# Abacus arithmetic

- Computational complexity of arithmetic is notation-dependent(!)
- For example,  $\pm$  in unary arithmetic is concatenation and decatenation
- Multiplication and division by natural powers of the radix is  $\mathcal{O}(1)$
- We can describe the abacus as a kind of abstract rewriting system



# Abacus dependent types

```
sealed class B<X, P : B<X, P>>(open val x: X? = null) {  
    val T: T<P> get() = T(this as P)  
    val F: F<P> get() = F(this as P)  
}
```

```
class U(val i: Int) : B<Any, U>() // Checked at runtime
```

```
object Ø: B<Ø, Ø>(null) // Denotes the end of a bitlist
```

```
class T<X>(override val x: X = Ø as X) : B<X, T<X>>(x)  
{ companion object: T<Ø>(Ø) }
```

```
class F<X>(override val x: X = Ø as X) : B<X, F<X>>(x)  
{ companion object: F<Ø>(Ø) }
```

```
val b0: F<Ø> = F
```

```
val b1: T<Ø> = T
```

```
val b2: F<T<Ø>> = T.F // Note the raw type is reversed
```

```
val b4: F<F<T<Ø>>> = T.F.F
```

# Abacus dependent types

```
typealias B_0<K> = F<K> // Type synonyms for legibility
typealias B_1<K> = T<K>
typealias B_2<K> = F<T<K>>
typealias B_3<K> = T<T<K>>
typealias B_4<K> = F<F<T<K>>>
typealias B_7<K> = T<T<T<K>>>
typealias B_8<K> = F<F<F<T<K>>>>
```

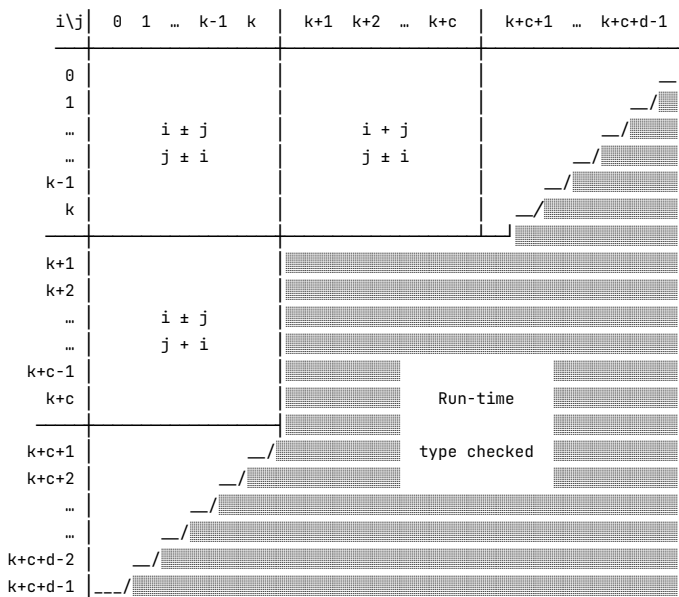
```
// Calculates  $k + 1$  for all  $k = 2^n - 1$ ,  $0 \leq n < 4$ 
```

```
operator fun Ø.plus(t: T<Ø>) = b1
operator fun B_0<Ø>.plus(t: T<Ø>) = b1
operator fun B_1<Ø>.plus(t: T<Ø>): B_2<Ø> = F(x + b1)
operator fun B_3<Ø>.plus(t: T<Ø>): B_4<Ø> = F(x + b1)
operator fun B_7<Ø>.plus(t: T<Ø>): B_8<Ø> = F(x + b1)
```

```
// Calculates  $k + 1$  for all  $k \equiv 2^n - 1 \pmod{2^{n+1}}$ ,  $1 \leq n < 4$ 
```

```
operator fun <K: B<*, *>> B_0<K>.plus(t: T<Ø>) = T(x)
operator fun <K: B<*, *>> B_1<F<K>>.plus(t: T<Ø>) = F(x + b1)
operator fun <K: B<*, *>> B_3<F<K>>.plus(t: T<Ø>) = F(x + b1)
operator fun <K: B<*, *>> B_7<F<K>>.plus(t: T<Ø>) = F(x + b1)
```

# Abacus dependent types: birds eye view



# Annotated history of typed eDSLs

- Canning et al. (1989) - F-Bounded Polymorphism is first invented
- Cheney & Hinze (2003) - Phantom types (good for type-safe builders)
- Meijer et al. (2006) - Language integrated querying (LINQ)
- Eder (2011) - Commercial reimplementations LINQ in Java/jOOQ
- Grigore (2016) - Java Generics shown to be Turing Complete
- Erdős (2017) - Encodes Boolean logic into Java type system
- Nakamaru et al. (2017) - Silverchain: a fluent API generator
- **Considine (2019) - Shape-safe matrix multiplication in Kotlin $\nabla$**
- Gil & Roth (2019) - Fling, a fluent API parser generator
- Cheng (2020) - Automatic theorem proving in the Scala type system
- Roth (2021) - Encodes CFL into Nominal Subtyping with Variance
- **Considine (2021) - Arithmetic in Kotlin via typelevel abacus**
- We know how to lower parsing onto types, what about vis versa?

# Can we lower type checking onto parsing?

First, let us consider the untyped version:

```
Exp -> 0 | 1 | ... | T | F
Exp -> Exp Op Exp | if ( Exp ) Exp else Exp
Op  -> and | or | + | *
```

Now, let us consider the GADT/HOAS version:

```
Exp<Bool> -> T | F
Op<Bool>  -> and | or
Exp<Int>  -> 0 | 1 | ... | 9
Op<Int>   -> + | *
Exp<E>    -> Exp<E> Op<E> Exp<E> // Es must be exactly the same!
Exp<E>    -> if ( Exp<Bool> ) Exp<E> else Exp<E>
```

We can eliminate contextuality by concretizing over  $E \rightarrow \text{Bool} \mid \text{Int}$ :

```
Exp<Bool> -> T | F
Exp<Bool> -> Exp<Bool> or Exp<Bool> | Exp<Bool> and Exp<Bool>
Exp<Bool> -> if ( Exp<Bool> ) Exp<Bool> else Exp<Bool>
Exp<Int>  -> 0 | 1 | ... | 9
Exp<Int>  -> Exp<Int> + Exp<Int> | Exp<Int> * Exp<Int>
Exp<Int>  -> if ( Exp<Bool> ) Exp<Int> else Exp<Int>
```

# Inductive and algebraic graph representations

We can represent a graph inductively, using a CFG/ADT:

$$\begin{aligned}\text{VERTEX} &\rightarrow \text{INT} \\ \text{NEIGHBORS} &\rightarrow \text{VERTEX} \mid \text{VERTEX NEIGHBORS} \\ \text{CONTEXT} &\rightarrow ([\text{NEIGHBORS}], \text{VERTEX}, [\text{NEIGHBORS}]) \\ \text{GRAPH} &\rightarrow \text{EMPTY} \mid \text{CONTEXT GRAPH}\end{aligned}$$

We can also represent graphs algebraically using the graph Laplacian:

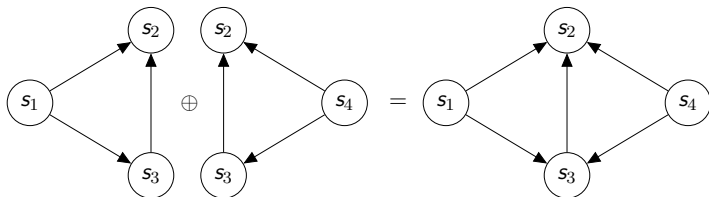
$$\mathcal{L}_{i,j} := \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } (v_i \rightarrow v_j) \in E \\ 0 & \text{otherwise,} \end{cases}$$

The latter form is preferred for representation learning [Hamilton (2020)].

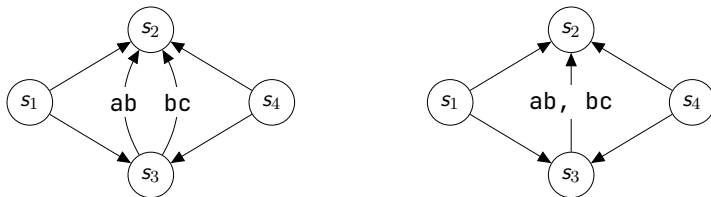


# Graph combinator

To merge two unlabeled graphs, apply  $G_1 \oplus G_2 = (V_1 \cup V_2) \times (E_1 \cup E_2)$ :



Can be specialized to join ADTs, e.g.:  $G_1 \oplus G_2 = (V_1 \cup V_2) \times (E_1 \bowtie E_2)$ :



# A type family for graphs

```
interface IGF<G, E, V> where
  G: IGraph<G, E, V>, E: IEdge<G, E, V>, V: IVertex<G, E, V> {
    val G: (vertices: Set<V>) -> G
    val E: (s: V, t: V) -> E
    val V: (old: V, edgeMap: (V) -> Set<E>) -> V

    fun G(vararg graphs: G): G = G(graphs.toList())
    fun G(vararg vertices: V): G = G(vertices.map { it.graph })
    fun G(l: List<Any>): G = when {
      l allAre G -> l.fold(G()) { it, acc -> it + acc as G }
      l allAre V -> list.map { it as V }.toSet()
    }.let { G(it) }

    operator fun G.plus(that: G): G =
      G((this - that) + (this join that) + (that - this))

    operator fun G.minus(that: G): G = G(vertices - that.vertices)

    infix fun G.join(that: G): Set<V> = TODO("Override me!")
  }
```

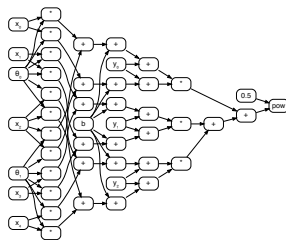
# Classical programs are graphs

Programs can be compiled into DFGs and represented using a big matrix.

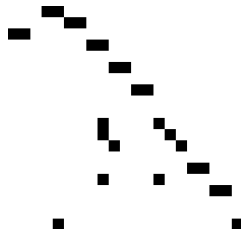
## Program

```
sum = 0
l = [0, 0, 0, 0]
for i in range(0, 4):
    l[i] += 0[i] * x[i]
for i in range(0, 4):
    l[i] -= y[i] - b
for i in range(0, 4):
    l[i] *= l[i]
for i in range(0, 4):
    sum += l[i]
l = sqrt(sum)
```

## Dataflow Graph



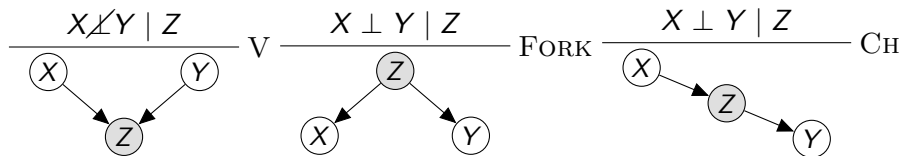
## Matrix



This representation allows us to solve for their fixedpoints as eigenvectors.

# Probabilistic programs are also graphs

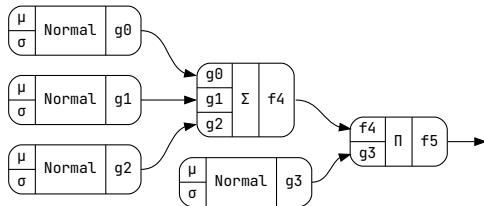
A Bayesian Belief Network (BN) is an acyclic DGM of the following form:



$$P(x_1, \dots, x_D) = \prod_{i=1}^D P(x_i \mid \text{parents}(x_i))$$

Translatable to a probabilistic circuit a.k.a. Sum Product Network (SPN):

$$\begin{aligned} PC &\rightarrow v \sim \mathcal{D} \\ PC &\rightarrow PC \oplus PC \\ PC &\rightarrow PC \otimes PC \end{aligned}$$



# Message passing & path algebras

A semiring algebra, denoted  $(S, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ , is a set together with two binary operators  $\oplus, \otimes : S \times S \rightarrow S$  such that  $(S, \oplus, \mathbb{0})$  is a commutative monoid and  $(S, \otimes, \mathbb{1})$  is a monoid. Furthermore, we have distributivity:

$$\frac{a \bullet (b \bullet c)}{(a \bullet b) \bullet c} \text{ ASSOC} \qquad \frac{a \bullet \mathbb{1}}{a} \text{ NEUTRAL} \qquad \frac{a \bullet b}{b \bullet a} \text{ COMM}$$

$$\frac{(a \oplus b) \otimes c}{(a \otimes c) \oplus (b \otimes c)} \text{ DIST} \qquad \frac{a \otimes \mathbb{0}}{\mathbb{0}} \text{ ANNHIL}$$

These operators can be lifted to matrices to form *path algebras*:

$$\delta_{st} = \overbrace{\bigoplus_{P \in P_{st}^*} \bigotimes_{e \in P} W_e}^{\text{Update}}$$

Aggregate

$\oplus$	$\otimes$	$\mathbb{0}$	$\mathbb{1}$	Path
min	+	$\infty$	0	Shortest
max	+	$-\infty$	0	Longest
max	min	0	$\infty$	Widest
$\underline{\vee}$	$\wedge$	$\circ$	$\top$	Random

# Linear Finite State Registers

Let  $\mathbf{M} : \text{GF}(2^{n \times n})$  be a square matrix  $\mathbf{M}_{r,c}^0 = P_c$  if  $r = 0$  else  $\mathbb{1}[c = r - 1]$ , where  $P$  is a feedback polynomial over  $\text{GF}(2^n)$  with coefficients  $P_{1\dots n}$  and semiring operators  $\oplus := \underline{\vee}, \otimes := \wedge$ :

$$\mathbf{M}^t \mathbf{V} = \begin{pmatrix} P_1 & P_2 & P_3 & P_4 & P_5 \\ \top & \circ & \circ & \circ & \circ \\ \circ & \top & \circ & \circ & \circ \\ \circ & \circ & \top & \circ & \circ \\ \circ & \circ & \circ & \top & \circ \end{pmatrix}^t \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{pmatrix}$$

Selecting any  $V \neq \mathbf{0}$  and coefficients  $P_j$  from a known *primitive polynomial* then powering the matrix  $\mathbf{M}$  generates an ergodic sequence over  $\text{GF}(2^n)$ :

$$\mathbf{S} = (V \quad \mathbf{M}V \quad \mathbf{M}^2V \quad \mathbf{M}^3V \quad \dots \quad \mathbf{M}^{2^n-1}V)$$

This sequence has *full periodicity*, i.e., for all  $i, j \in [0, 2^n)$ ,  $\mathbf{S}_i = \mathbf{S}_j \Rightarrow i = j$ .

# Linear finite state registers

a	b	c	d	e		$V_1$
1	0	0	0	0		$V_2$
0	1	0	0	0	*	$V_3$
0	0	1	0	0		$V_4$
0	0	0	1	0		$V_5$

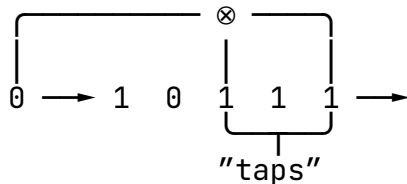
$$M \quad * \quad V$$

$$S_4 = M \dots S_1 = M * V$$

11	11	11
0	0	1
0	1	0
1	0	1
0	1	1
1	0	1

$$P = 20 = 1 + x^3 + x^5$$

1	0	1	0	1
11	11	11	11	11
a	b	c	d	e



$$S_0 = 1 \quad 0 \quad 1 \quad 1 \quad 1$$

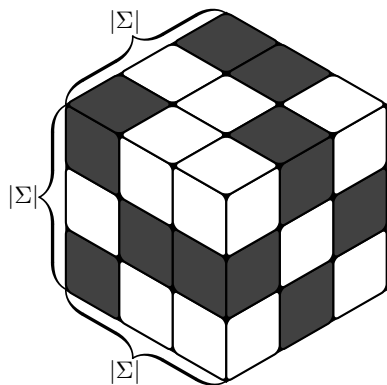
$$S_1 = 0 \quad 1 \quad 0 \quad 1 \quad 1$$

$$S_2 = 1 \quad 0 \quad 1 \quad 0 \quad 1$$

$$S_3 = 0 \quad 1 \quad 0 \quad 1 \quad 0$$

$$S_4 = 0 \quad 0 \quad 1 \quad 0 \quad 1$$

# Multidimensional sampling: the hasty pudding trick



To uniformly sample  $\sigma \sim \Sigma^n$  without replacement, we could track historical samples, or, we can form an injection  $GF(2^n) \rightarrow \Sigma^d$ , cycle a primitive polynomial over  $GF(2^n)$ , then discard samples that do not identify an element in any indexed dimension. This procedure rejects  $(1 - |\Sigma|2^{-\lceil \log_2 |\Sigma| \rceil})^d$  samples on average and requires  $\sim \mathcal{O}(1)$ .

e.g.,  $\Sigma^2 = \{A, B, C\}^2$ ,  $x^4 + x^3 + 1$

$S_0$	$S_1$	$S_2$	$S_3$	$S_4$	$S_5$	$S_6$	$S_7$
$\begin{smallmatrix}    \\ [1000] \end{smallmatrix}$	$\begin{smallmatrix}    \\ [0100] \end{smallmatrix}$	$\begin{smallmatrix}    \\ [0010] \end{smallmatrix}$	$\begin{smallmatrix}    \\ [1001] \end{smallmatrix}$	$\begin{smallmatrix}    \\ [1100] \end{smallmatrix}$	$\begin{smallmatrix}    \\ [0110] \end{smallmatrix}$	$\begin{smallmatrix}    \\ [1011] \end{smallmatrix}$	$\begin{smallmatrix}    \\ [0101] \end{smallmatrix}$
C A	B A	A C	C B		B C		B B



# Multidimensional no-replacement sampler

```
fun List<Int>.bitLens() = map { ceil(log2(it.toDouble())).toInt() }

// Splits a bitvector into designated chunks and returns indices
// (10101011, [3, 2, 3]) -> [101, 01, 011] -> [4, 1, 3]
fun List<Boolean>.toIndexes(bitLens: List<Int>): List<Int> =
    bitLens.fold(listOf<List<Boolean>>()) to this { (a, b), i ->
        (a + listOf(b.take(i))) to b.drop(i)
    }.first.map { it.toInt() }

fun Sequence<List<Boolean>>.hastyPudding(lengths: List<Int>) =
    map { it.toIndexes(lengths.bitLens()) }
    .filter { it.zip(lengths).all { (a, b) -> a < b } }

fun <T> List<Set<T>>.sampleWithoutReplacement(
    lengths: List<Int> = map { it.size },
    bitLens: List<Int> = map(Set<T>::size).bitLens(),
    degree: Int = bitLens.sum().also { println("LFSR(GF(2^$it))") }
): Sequence<List<T>> =
    LFSR(degree).hastyPudding(lengths)
    .map { zip(it).map { (dims, idx) -> dims[idx] } }
```

# Recap: Classical logic in a nutshell

$$\frac{a \vee b}{(p \vee q) \wedge \neg(p \wedge q)} \text{ XOR} \qquad \frac{a \rightarrow b}{\neg a \vee b} \text{ Impl}$$
$$\frac{a \leftrightarrow b}{(\neg a \vee b) \wedge (\neg b \vee a)} \text{ Iff}$$

$$\frac{\neg \neg a}{a} \text{ 2Neg}$$

$$\frac{a \bullet (b \bullet c)}{(a \bullet b) \bullet c} \text{ Assoc}_{\wedge \vee}$$

$$\frac{a \bullet b}{b \bullet a} \text{ Comm}_{\wedge \vee}$$

$$\frac{a \wedge (b \vee c)}{(a \wedge b) \vee (a \wedge c)} \text{ Dist}_{\wedge}$$

$$\frac{a \vee (b \wedge c)}{(a \vee b) \wedge (a \vee c)} \text{ Dist}_{\vee}$$

$$\frac{\neg(a \vee b)}{\neg a \wedge \neg b} \text{ DeMorgan}_{\vee}$$

$$\frac{\neg(a \wedge b)}{\neg a \vee \neg b} \text{ DeMorgan}_{\wedge}$$

## Conjunctive Normal Form

CONJ  $\rightarrow$  (DISJ) | CONJ  $\wedge$  (DISJ)

UNIT  $\rightarrow$  VAR |  $\neg$ VAR |  $\perp$  |  $\top$

DISJ  $\rightarrow$  UNIT | DISJ  $\vee$  DISJ

$$\begin{array}{c} \frac{\neg(x \vee \neg y) \vee \neg \neg z}{\neg(x \vee \neg y) \vee z} \text{2Neg} \\ \frac{\neg(x \vee \neg y) \vee z}{(\neg x \wedge \neg \neg y) \vee z} \text{DeMorgan} \\ \frac{(\neg x \wedge \neg \neg y) \vee z}{(\neg x \wedge y) \vee z} \text{2Neg} \\ \frac{(\neg x \wedge y) \vee z}{(\neg x \vee z) \wedge (y \vee z)} \text{Dist} \end{array}$$

## Zhegalkin Normal Form

$$f(x_1, \dots, x_n) = \bigoplus_{i \subseteq \{1, \dots, n\}} a_i x^i$$

i.e.,  $a_i$ 's filter the powerset.

$$\begin{array}{c} \frac{x + (y \wedge \neg z)}{x + y(1 \oplus z)} \\ \frac{x + y(1 \oplus z)}{x + (y \oplus yz)} \\ \frac{x \oplus (y \oplus yz) \oplus x(y \oplus yz)}{x \oplus y \oplus xy \oplus yz \oplus xyz} \end{array}$$

# Some common algebraic and logical forms

$a_1$	$a_2$	$a_3$	$a_4$	ZNF	Logical	CNF
0	0	0	0	0	$\perp$	$x \wedge \neg x$
1	0	0	0	1	$\top$	$x \vee \neg x$
0	1	0	0	$x$	$x$	$x$
1	1	0	0	$1 + x$	$\neg x$	$\neg x$
0	0	1	0	$y$	$y$	$y$
1	0	1	0	$1 + y$	$\neg y$	$\neg y$
0	1	1	0	$x + y$	$x \oplus y$	$(x \vee y) \wedge (\neg x \vee \neg y)$
1	1	1	0	$1 + x + y$	$x \iff y$	$(x \vee \neg y) \wedge (\neg x \vee y)$
0	0	0	1	$xy$	$x \wedge y$	$x \wedge y$
1	0	0	1	$1 + xy$	$\neg(x \wedge y)$	$(\neg x) \vee (\neg y)$
0	1	0	1	$x + xy$	$x \wedge (\neg y)$	$x \wedge (\neg y)$
1	1	0	1	$1 + x + xy$	$x \implies y$	$(\neg x) \vee y$
0	0	1	1	$y + xy$	$(\neg x) \wedge y$	$(\neg x) \wedge y$
1	0	1	1	$1 + y + xy$	$x \longleftarrow y$	$x \vee (\neg y)$
0	1	1	1	$x + y + xy$	$x \vee y$	$x \vee y$
1	1	1	1	$1 + x + y + xy$	$\neg(x \vee y)$	$(\neg x) \wedge (\neg y)$

# Facts about finite fields

- For every prime number  $p$  and positive integer  $n$ , there exists a finite field with  $p^n$  elements, denoted  $GF(p^n)$ ,  $\mathbb{Z}/p^n$  or  $\mathbb{F}_p^n$ .
- The following instruction sets have identical expressivity:
  - Pairs:  $\{\vee, \neg\}$ ,  $\{\wedge, \neg\}$ ,  $\{\rightarrow, \neg\}$ ,  $\{\rightarrow, \perp\}$ ,  $\{\rightarrow, \underline{\vee}\}$ ,  $\{\wedge, \underline{\vee}\}$ ,  $\dots$
  - Triples:  $\{\vee, =, \underline{\vee}\}$ ,  $\{\vee, \underline{\vee}, \top\}$ ,  $\{\wedge, =, \perp\}$ ,  $\{\wedge, =, \underline{\vee}\}$ ,  $\{\wedge, \underline{\vee}, \top\}$ ,  $\dots$
- In other words, we can compute any Boolean function  $\mathbb{B}^n \rightarrow \mathbb{B}$  by composing any one of the above operator sets in an orderly fashion.
- $\mathbb{F}_2$  corresponds to arithmetic modulo 2, i.e.,  $\oplus := \underline{\vee}$ ,  $\otimes := \wedge$ .
- There are (at least) two schools of thought about Boolean circuits:
  - Logical: Conjunctive Normal Form (CNF). May not be unique.
  - Algebra: Zhegalkin Normal Form (ZNF). Always unique.
- The type  $\mathbb{F}_2^n \rightarrow \mathbb{F}_2$  possesses  $2^{2^n}$  inhabitants.

# Preface to “Two Memoirs on Pure Analysis”

*“Long algebraic calculations were at first hardly necessary for mathematical progress... It was only since Euler that concision has become indispensable to continuing the work this great geometer has given to science. Since Euler, calculation has become more and more necessary and... the algorithms so complicated that progress would be nearly impossible without the elegance that modern geometers have brought to bear on their research, and by which means the mind can promptly and with a glance grasp a large number of operations.*

...

*It is clear that elegance, so admirably and aptly named, has no other purpose.*

...

*Jump headlong into the calculations! Group the operations, classify them by their difficulties and not their appearances. This, I believe, is the mission of future geometers. This is the road on which I am embarking in this work.”*

---

Évariste Galois, 1811-1832

# What's the point?

- Algebraists have developed a powerful language for rootfinding
- Tradition handed down from Euler, Galois, Borel, Kleene, Chomsky
- We know closed forms for exponentials of structured matrices
- Solving these forms can be much faster than power iteration
- Unifies many problems in PL, probability and graph theory
- Context-free parsing is just rootfinding on a semiring algebra
- Type checking sans recursive types is just graph reachability
- Unification/simplification is lazy hypergraph search
- Bounded program synthesis is matrix factorization/completion
- By doing so, we can leverage well-known algebraic techniques

## Parsing

- The line between parsing and computation is blurry
- Investigate connection between dynamical and term rewrite systems
- Extend Valiant's parser to tensors/context-sensitive languages
- Recover the original parse tree or eliminate Chomsky Normal Form
- What is the connection to Leibnizian differentiability?

## Probability

- Look into Markov chains (detailed balance, stationarity, reversibility)
- Fuse Valiant parser and probabilistic context-free grammar
- Message passing and graph diffusion processes
- Look into constrained optimization (e.g., L/QP) to rank feasible set



# Special thanks

Nghi D. Q. Bui  
Zhixin Xiong  
Brigitte Pientka  
David Yu-Tung Hui  
Ori Roth



**McGill**  
UNIVERSITY



Learn more at:

<http://oplss22.ndan.co>