langcc

A Next-Generation Compiler Compiler

Joe Zimmerman

`jzimmerman.io`

langcc

- Specify a language in declarative BNF syntax

```
Expr.BinOp1 <- x:Expr _ op:(Add:`+` | Sub:`-`) _ y:Expr;
Expr.BinOp2 <- x:Expr _ op:(Mul:`*` | Div:`/`) _ y:Expr;
```

- langcc generates a full compiler frontend
    - AST, lexer, parser, pretty-printer, traversals

langcc

- Can serve as replacement for lex+yacc, but much more powerful
  - Efficient, linear-time parsers for very general class of grammars
  - General enough for real industrial languages
    - Go 1.17.8: 1.2x faster than standard parser
    - Python 3.9.12: 4.3x faster than standard (CPython) parser
  - Generates full frontend (no more sprinkling C++ into your grammar!)

langcc

- Can serve as replacement for lex+yacc, but much more powerful
  - “Conflict tracing” algorithm: trace LR conflicts back to explicit “confusing input pair” (not opaque shift/reduce errors as in yacc)
  - Standalone “datatype compiler” generates C++ definitions of algebraic datatypes (including sum types)
  - langcc is *self-hosting*: generates its own frontend from BNF spec for “language of languages”

langcc

- Powered by novel innovations on canonical LR parsing

    1. Optimized LR NFA (LALR too restrictive; LR(k) can be efficient)
    2. *Attribute specs* unify precedence, associativity, semantic properties
    3. New grammar transformation: “CPS”
    4. Intuitive automata via recursive-descent actions (RD)
    5. Further extension: XLR (parallel LR parsing)
    6. Conflict tracing

langcc

- Powered by novel innovations on canonical LR parsing

1. **Optimized LR NFA (LALR too restrictive; LR(k) can be efficient)**
2. *Attribute specs* unify precedence, associativity, semantic properties
3. New grammar transformation: “CPS”
4. Intuitive automata via recursive-descent actions (RD)
5. Further extension: XLR (parallel LR parsing)
6. Conflict tracing

1. LR(k) NFA Optimization

- LR(k) often viewed as impractical due to size of LR NFA
  - Naively, requires a vertex for every pair [$X \rightarrow \alpha \,.\, \beta$, λ] where "$X \rightarrow \alpha \,.\, \beta$" is a dotted production and λ is a k-token follow string
  - Existing optimizations examine LR DFA and attempt to merge states without creating conflicts [Pag77, DM10]
  - We take a different approach:
    - Start with *LR(0) NFA*, determine a *partition* of follow strings for each vertex. Form modified LR(k) NFA based on these partitions.

1. LR(k) NFA Optimization

- LR(k) often viewed as impractical due to size of LR NFA
    - Naively, requires a vertex for every pair [$X \rightarrow \alpha . \beta , \lambda$] where "$X \rightarrow \alpha . \beta$" is a dotted production and λ is a k-token follow string

      (For clarity, we distinguish between "follow strings" and "lookaheads".)
    - Existing optimizations examine LR DFA and attempt to merge states without creating conflicts [Pag77, DM10]
    - We take a different approach:
        - Start with *LR(0) NFA*, determine a *partition* of follow strings for each vertex. Form modified LR(k) NFA based on these partitions.

1. LR(k) NFA Optimization

- Consider the following grammar:

$$S \rightarrow a\,X\,a \mid b\,X\,b \mid a\,Y\,b \mid b\,Y\,a$$

$$X \rightarrow c\,X'$$

$$Y \rightarrow c\,Y'$$

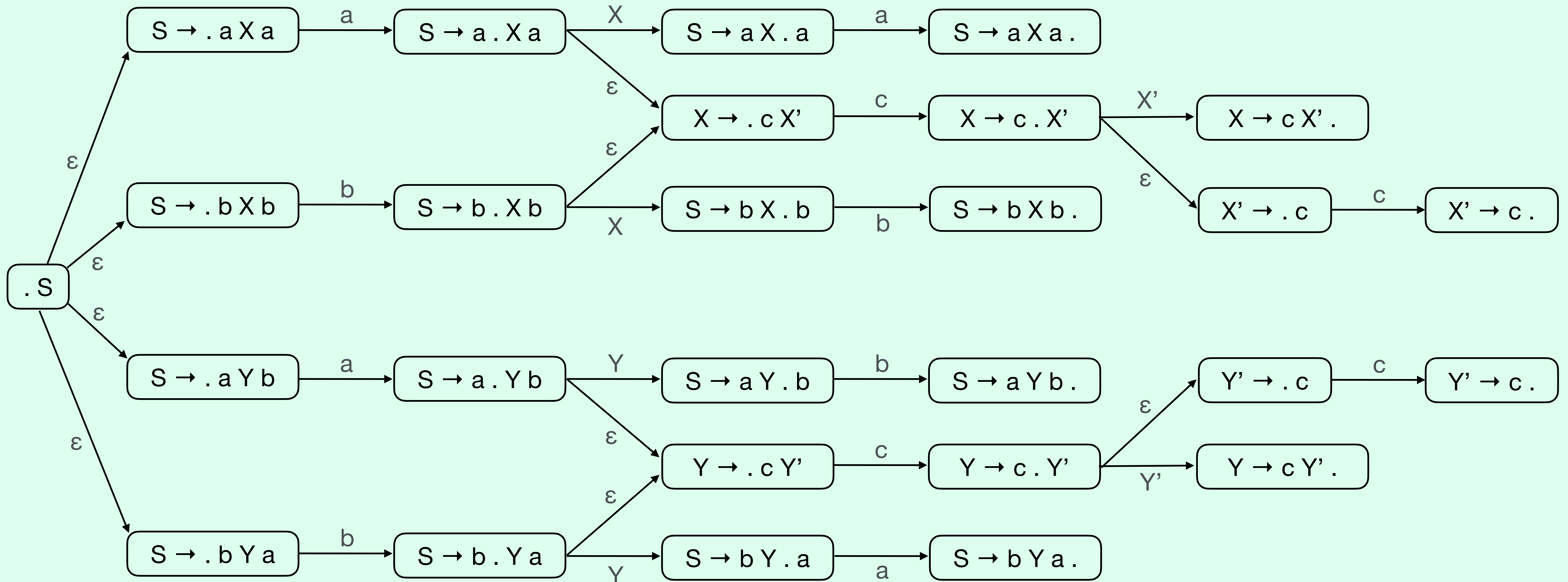
$$X' \rightarrow c$$

$$Y' \rightarrow c$$

1. LR(k) NFA Optimization

S → a X a | b X b | a Y b | b Y a
X → c X'   X' → c
Y → c Y'   Y' → c

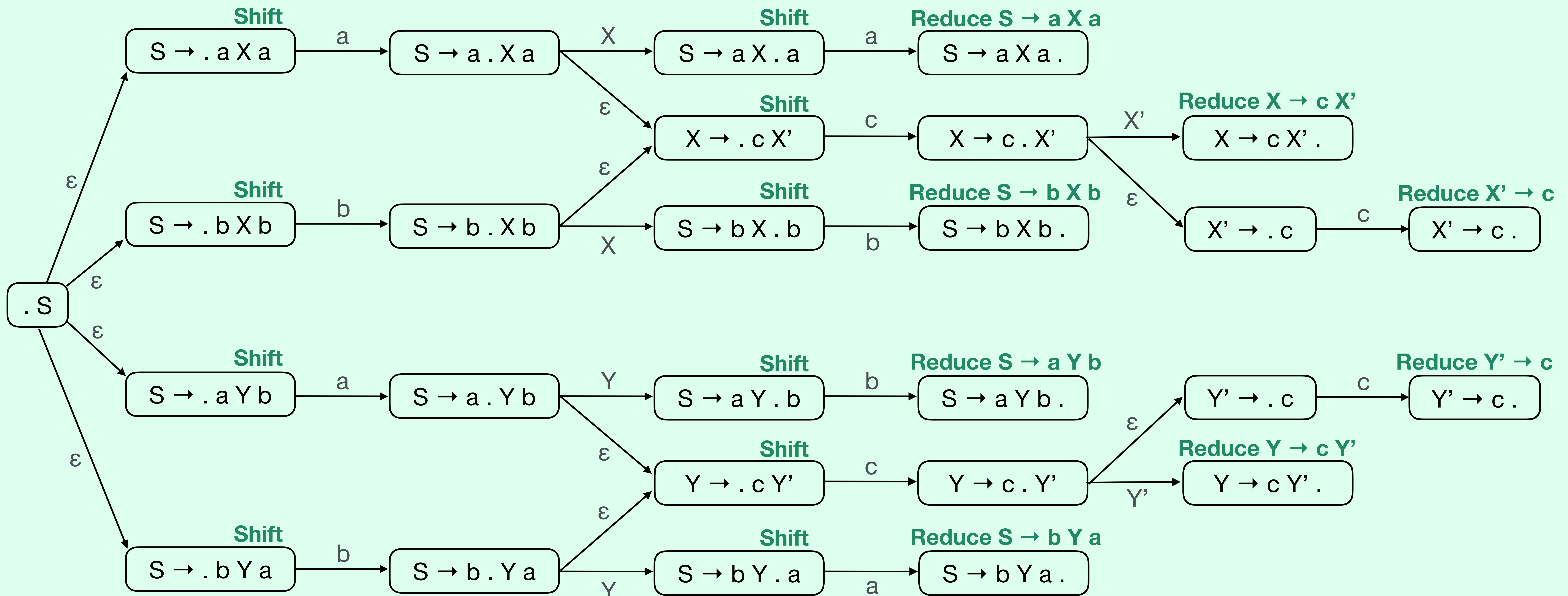
- The LR(0) NFA is as follows:



1. LR(k) NFA Optimization

S → a X a | b X b | a Y b | b Y a
X → c X'   X' → c
Y → c Y'   Y' → c

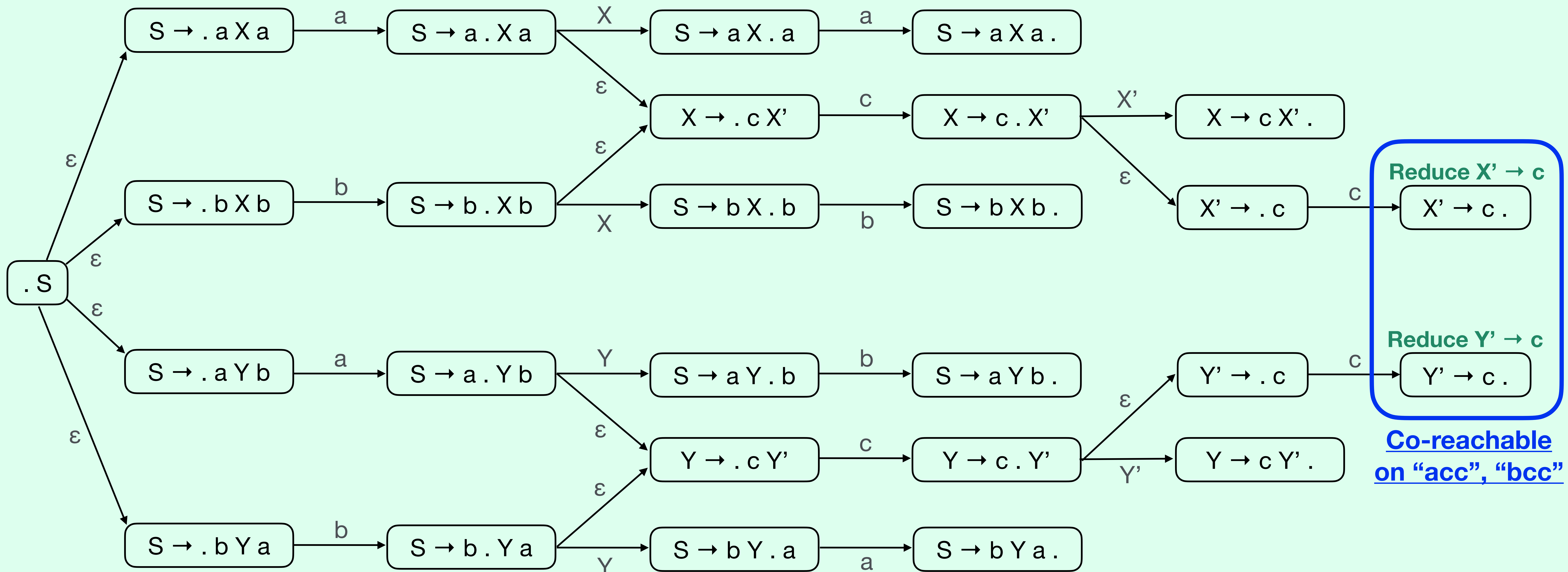
- The LR(0) NFA is as follows:



1. LR(k) NFA Optimization

S → a X a | b X b | a Y b | b Y a
X → c X'   X' → c
Y → c Y'   Y' → c

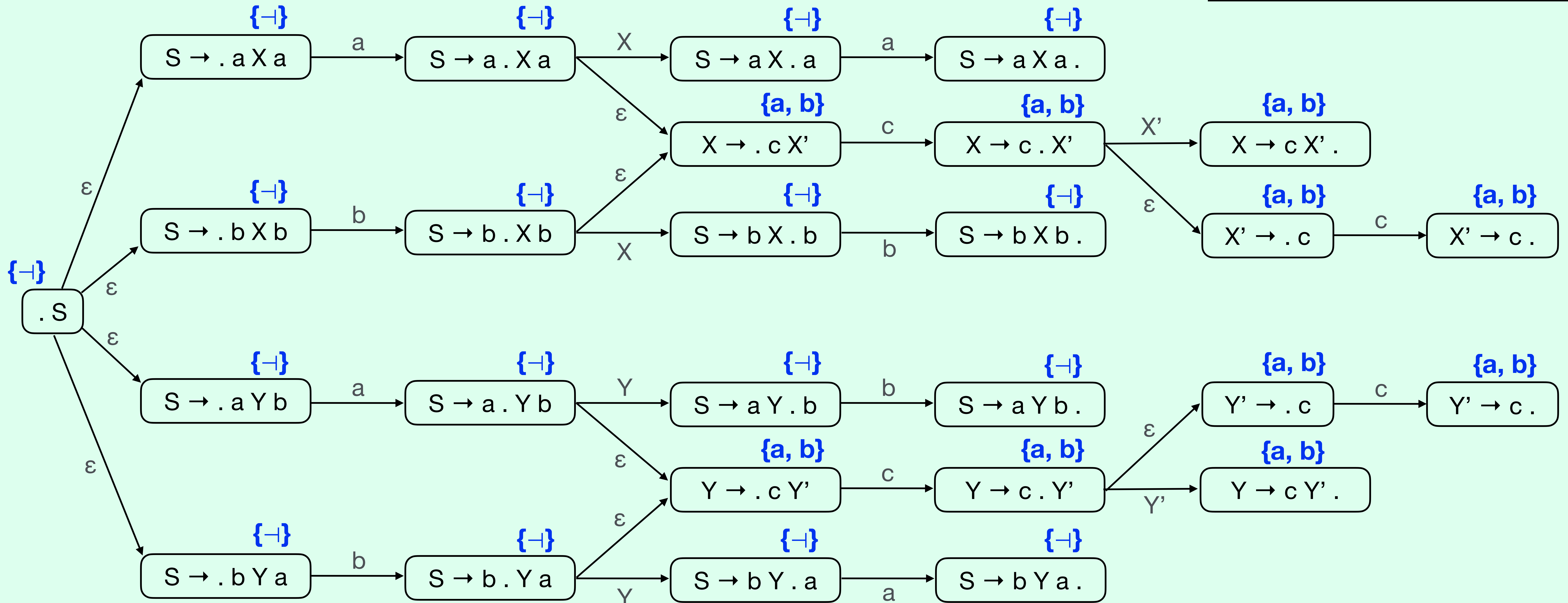
- The LR(0) NFA is as follows:



1. LR(k) NFA Optimization

- First, propagate reachable follow strings

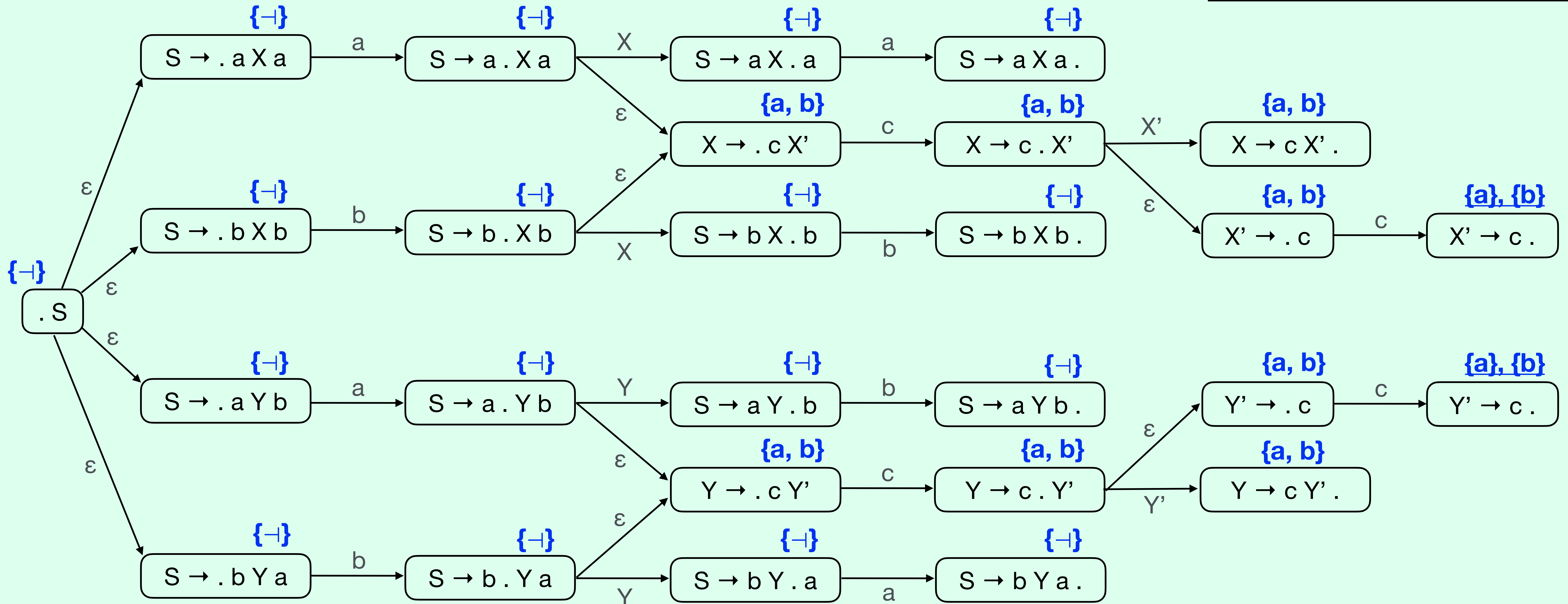
S → a X a | b X b | a Y b | b Y a
X → c X'   X' → c
Y → c Y'   Y' → c



1. LR(k) NFA Optimization

- Then, shatter potentially-conflicting vertices' partitions

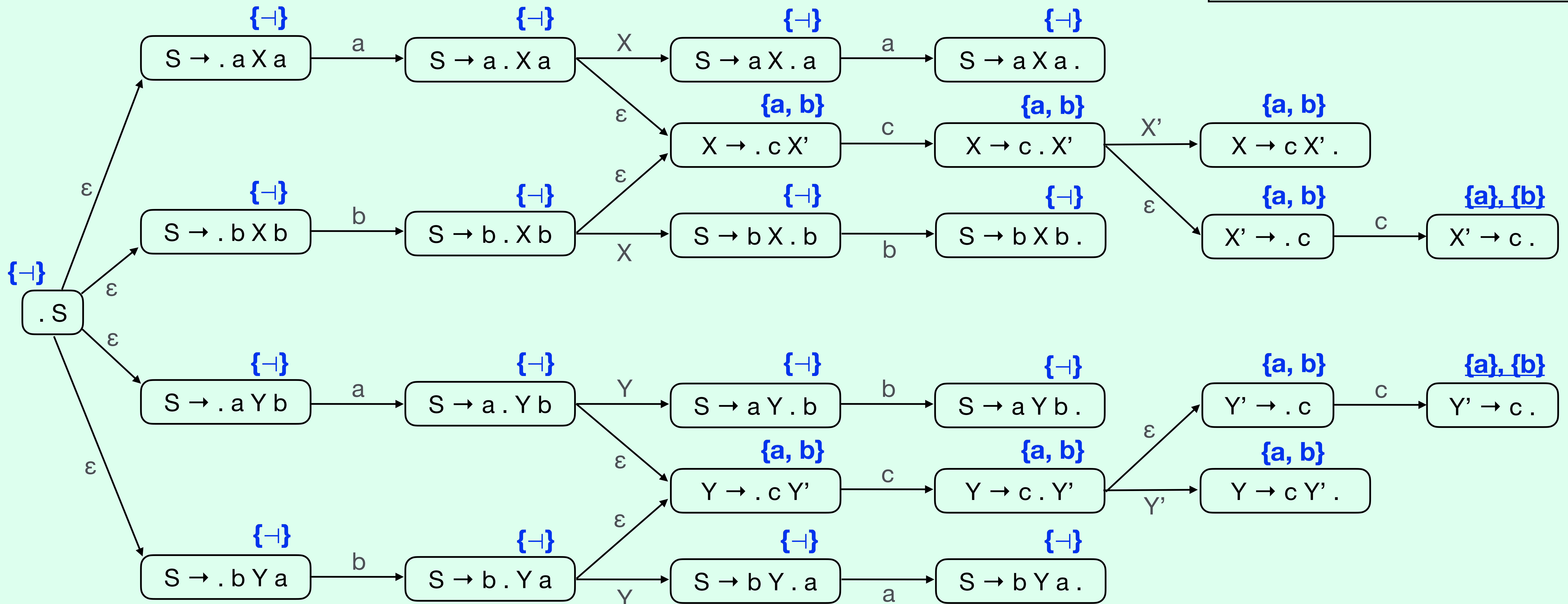
S → a X a | b X b | a Y b | b Y a
X → c X'   X' → c
Y → c Y'   Y' → c



1. LR(k) NFA Optimization

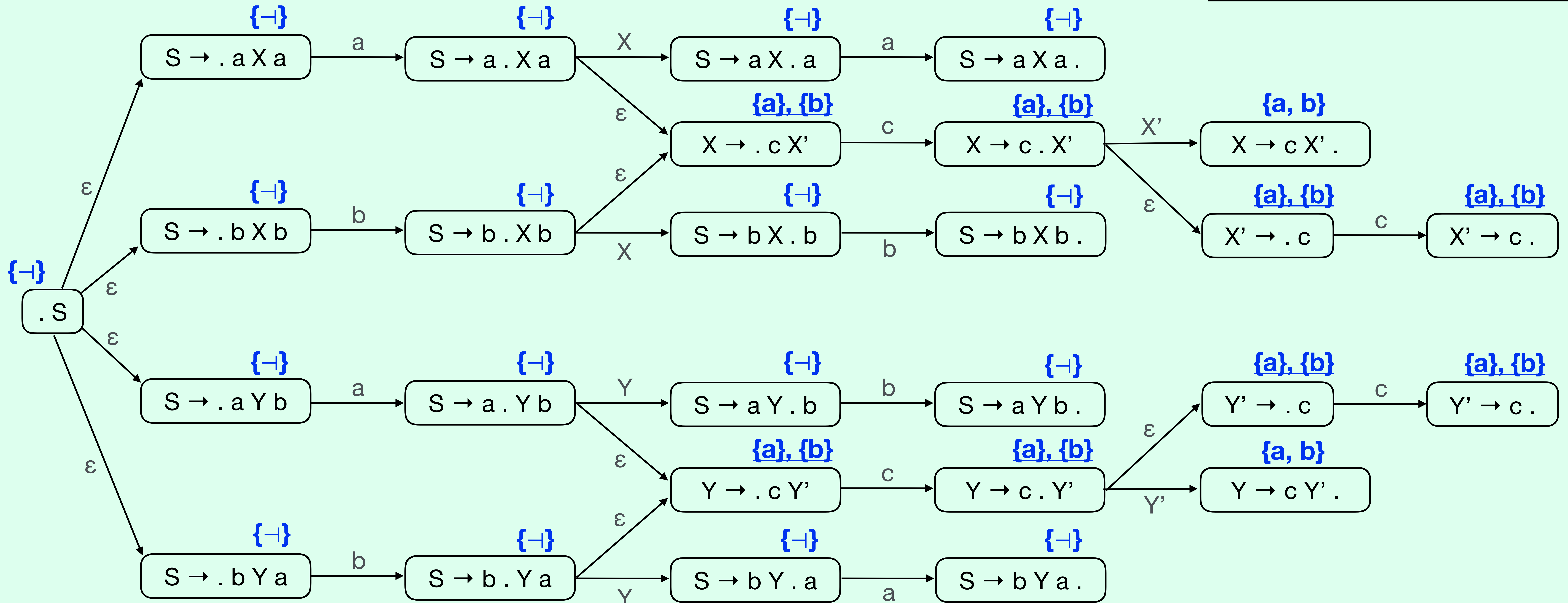
- Then, propagate partitions backward until fixed point

S → a X a | b X b | a Y b | b Y a
X → c X'   X' → c
Y → c Y'   Y' → c



1. LR(k) NFA Optimization

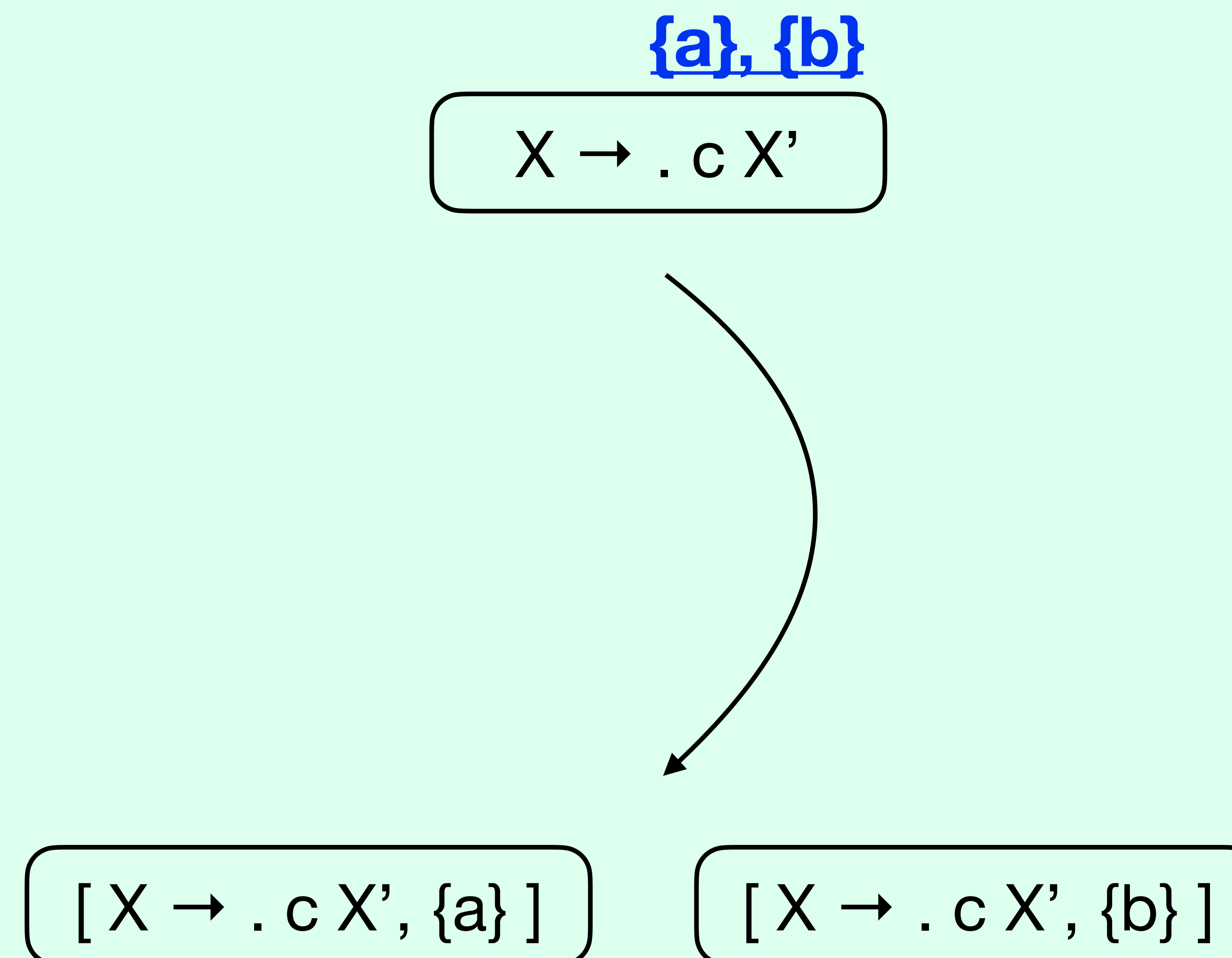
- Then, propagate partitions backward until fixed point



1. LR(k) NFA Optimization

- Final partitions determine LR(k) vertices

S → a X a | b X b | a Y b | b Y a	
X → c X’	X’ → c
Y → c Y’	Y’ → c



1. LR(k) NFA Optimization

- Back-propagation from potential conflicts yields partitions of follow sets at each node of LR(0) NFA
- Use canonical LR(k) NFA construction (operating on subsets of follow sets)
- Standard subset construction then produces LR(k) DFA, standard stack-based LR parsing algorithm
- In practice, *much* smaller than canonical LR(k) DFA

1. LR(k) NFA Optimization

- We have just described the *backward phase* of the LR(k) NFA optimization procedure

- In full procedure, there is also a *forward phase* in which partitions are propagated forward from the starting vertex

  - See accompanying paper “Practical LR Parser Generation” [Zim22] for details

langcc

- Powered by novel innovations on canonical LR parsing

    1. Optimized LR NFA (LALR too restrictive; LR(k) can be efficient)
    2. ***Attribute specs*** **unify precedence, associativity, semantic properties**
    3. New grammar transformation: “CPS”
    4. Intuitive automata via recursive-descent actions (RD)
    5. Further extension: XLR (parallel LR parsing)
    6. Conflict tracing

2. Attribute specs

- Many real language constructs are not LR(k)
  - Precedence/associativity: $E \rightarrow E + E \mid E * E \mid x$
    - Can be hacked around, but want a declarative solution that does not modify grammar, and does not operate at level of LR conflicts
  - Semantic attributes (e.g., “expr in value context”, “expr in type context”)
  - *Syntactic* attributes
    - Golang: `if * func () {}`

2. Attribute specs

- langcc provides a unified solution: *attribute specs*
- Attributes can be either binary-valued {0, 1}, or integer-valued {0, …, n}
  - Binary-valued typically used for semantic properties (e.g., “type expr”)
  - Integer-valued typically used for precedence levels
- Grammar rules accompanied by *attribute constraints*

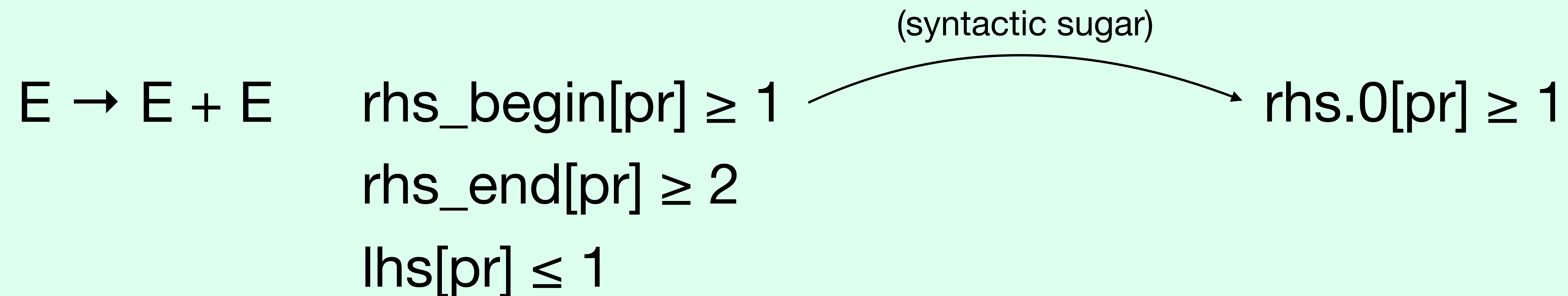
2. Attribute specs

- Example: operator precedence

$E \rightarrow E + E$	rhs_begin[pr] ≥ 1 rhs_end[pr] ≥ 2 lhs[pr] ≤ 1
$E \rightarrow E * E$	rhs_begin[pr] ≥ 2 rhs_end[pr] ≥ 3 lhs[pr] ≤ 2
$E \rightarrow x$	lhs[pr] ≤ 3

2. Attribute specs

- Example: operator precedence



E → E * E   rhs_begin[pr] ≥ 2
rhs_end[pr] ≥ 3
lhs[pr] ≤ 2

E → x   lhs[pr] ≤ 3

2. Attribute specs

- Attribute constraints carried through to NFA edges
    - Avoid blowing up grammar with extra productions (Cartesian product)
    - Avoid changing semantics of AST
- Attribute constraints required to be *separable and monotonic:*

$$lhs[attrA] \leq C$$
$$rhs.i[attrA] \geq C$$
$$lhs[attrA] \leq rhs.i[attrB]$$

2. Attribute specs

- Additional phase transforms precedence ordering into corresponding attribute constraints

    - N.B.: In full implementation, precedence uses "left-precedence" and "right-precedence" attributes (prL, prR) which enable prefix/postfix operators as well as infix.

```
f.g()     f().g
```

- Binary-valued attribute constraints expressed directly by user

    - For negated attributes, can use "attrNotA" in place of "¬attrA"

    - "

langcc

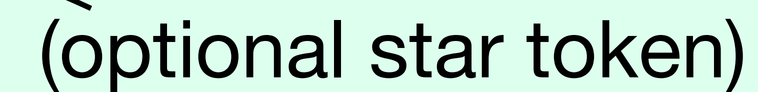
- Powered by novel innovations on canonical LR parsing

1. Optimized LR NFA (LALR too restrictive; LR(k) can be efficient)
2. *Attribute specs* unify precedence, associativity, semantic properties
3. **New grammar transformation: “CPS”**
4. Intuitive automata via recursive-descent actions (RD)
5. Further extension: XLR (parallel LR parsing)
6. Conflict tracing

3. CPS

- Some real language constructs are non-LR(k) for “incidental reasons”
- Golang BNF (simplified):

  Field.Embedded → *? E



  Field.Standard → E E

- Flattened to CFG:

  Field → X E

  X → *

  X → ε

  Field → E E

Suppose we are parsing a Field, and we see the beginning of an expression E…

Must decide whether to reduce X → ε immediately!

3. CPS

- Intuitively, we would like to “wait until the end of the BNF rule” to decide
- New grammar transformation (CPS, “continuation-passing style”) does precisely this

Field → X E
X → *
X → ε
Field → E E

CPS

Field → X’
X’ → * E
X’ → E
Field → E E

3. CPS

- CPS-transformed grammar remains linear in size of original grammar
- Automatic parse-time transformation to produce same AST as original

$S \rightarrow (a \mid b)\ (c \mid d)\ (e \mid f)$

$S \rightarrow X_0\, X_1\, X_2$
$X_0 \rightarrow a$
$X_0 \rightarrow b$
$X_1 \rightarrow c$
$X_1 \rightarrow d$
$X_2 \rightarrow e$
$X_2 \rightarrow f$

CPS

$S \rightarrow Y_0$
$Y_0 \rightarrow a\, Y_1$
$Y_0 \rightarrow b\, Y_1$
$Y_1 \rightarrow c\, Y_2$
$Y_1 \rightarrow d\, Y_2$
$Y_2 \rightarrow e$
$Y_2 \rightarrow f$

langcc

- Powered by novel innovations on canonical LR parsing

    1. Optimized LR NFA (LALR too restrictive; LR(k) can be efficient)

    2. *Attribute specs* unify precedence, associativity, semantic properties

    3. New grammar transformation: “CPS”

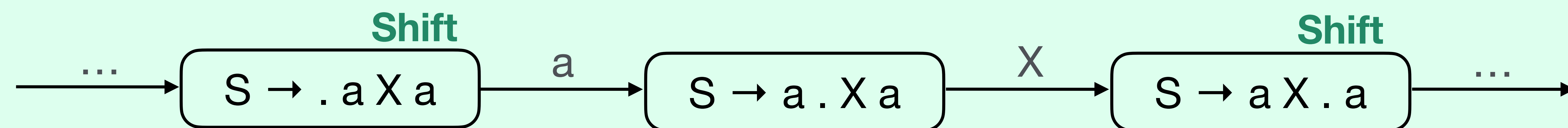
    4. **Intuitive automata via recursive-descent actions (RD)**

    5. Further extension: XLR (parallel LR parsing)

    6. Conflict tracing

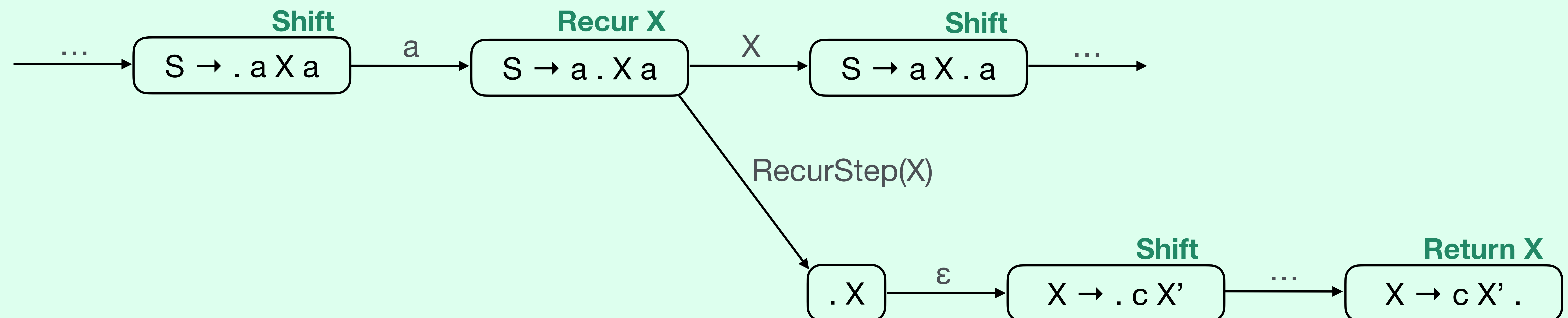
4. Recursive-descent (RD) actions

- Recursive-descent parsers have intuitive runtime behavior
  - “Parsing a program” → “parsing a statement” → “parsing an expression”
- Would like similar properties with bottom-up (LR-style) parsing
- Solution: *recursive-descent (RD) actions*



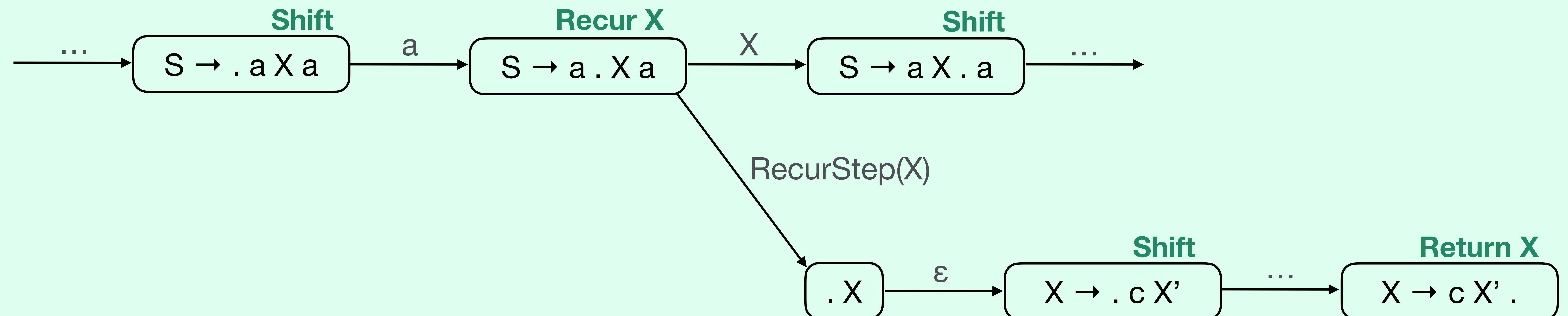
4. Recursive-descent (RD) actions

- Recursive-descent parsers have intuitive runtime behavior
  - “Parsing a program” → “parsing a statement” → “parsing an expression”
- Would like similar properties with bottom-up (LR-style) parsing
- Solution: *recursive-descent (RD) actions*



4. Recursive-descent (RD) actions

- Enables much more intuitive (and often smaller) LR automata
- May cause additional conflicts (“Recur”, “Return” as well as “Shift”, Reduce”)
    - But can easily “opt out” of RD for any given nonterminal, by specifying that it is to be “unfolded” by langcc



langcc

- Powered by novel innovations on canonical LR parsing

1. Optimized LR NFA (LALR too restrictive; LR(k) can be efficient)
2. *Attribute specs* unify precedence, associativity, semantic properties
3. New grammar transformation: “CPS”
4. Intuitive automata via recursive-descent actions (RD)
5. **Further extension: XLR (parallel LR parsing)**
6. Conflict tracing

5. XLR

- Further theoretical refinement of LR paradigm:
    - A grammar G is XLR(k, t) (“extended LR”) if for every string x and every lookahead λ, x has at most t partial LR parses consistent with λ.
- Results (further details in paper):
    - If G is XLR(k, t) for fixed k and t, then G can be parsed in linear time.
    - There is a simple heuristic (based on “fork points”) to analyze a given LR NFA and determine an XLR(k, t) bound.

5. XLR

- XLR differs from Generalized LR parsing (GLR) in that GLR memoizes intermediate results.
    - GLR runs in polynomial time on all grammars G, while XLR may require exponential time on grammars that are not XLR(k, t)…
    - But if G is XLR(k, t) then XLR is highly efficient (in constant factors).
- N.B.: XLR is not implemented in langcc
    - We believe XLR should only be used as a last resort
    - If a grammar requires XLR, it is already “confusing” to a human reader!

langcc

- Powered by novel innovations on canonical LR parsing

    1. Optimized LR NFA (LALR too restrictive; LR(k) can be efficient)
    2. *Attribute specs* unify precedence, associativity, semantic properties
    3. New grammar transformation: “CPS”
    4. Intuitive automata via recursive-descent actions (RD)
    5. Further extension: XLR (parallel LR parsing)
    6. **<u>Conflict tracing</u>**

6. Conflict tracing

- On LR conflicts, existing tools (e.g., yacc) give opaque shift/reduce errors
- langcc instead provides an efficient “conflict tracing” algorithm
    - Traces an LR conflict back to an explicit, short “confusing input pair”

```
===== LR conflict 1 of 1

    &Expr                  &Expr
                 RecurStep(Expr)
  X0=(`-`)                   `-`
                 RecurStep(Expr)
     Expr                     id

                       Ret(Expr)    Shift

                             `+`    `+`
                              id    id
```

6. Conflict tracing

- Algorithm sketch:
  - For each LR DFA vertex with conflicting actions on some lookahead:
    - Perform a shortest-path search to find a sequence of symbols that leads to that vertex (“shortest” in the minimal generated strings of those symbols).
    - For each conflicting lookahead, and for each corresponding NFA vertex:
      - Perform a reverse shortest-path search in the NFA from that vertex to the starting vertex, concatenating the remaining tails of all visited productions (“shortest” in the minimal generated strings matching the given partial lookahead).
  - For the representative conflict of each conflicting action set, take the item from the previous loop that produces the shortest total input length.

langcc

- langcc is a next-generation compiler compiler
    - Powered by novel innovations on canonical LR parsing
    - Generates a full compiler frontend from declarative BNF input
    - Efficient, intuitive implementation with easy debugging of conflicts

Claim:

If your grammar is easy-to-parse for a human, langcc will generate your frontend automatically.

langcc

- So you want to write a new language…
  - And you are thinking of using lex+yacc…
    - Try langcc instead — it's powerful, efficient, easy to debug, and generates a full frontend!
  - And you are thinking of writing a recursive-descent parser by hand, because you know lex+yacc won't support your grammar…
    - Try langcc instead — it supports a very general class of grammars!

langcc.io