

Disambiguation and Layout-Sensitive Syntax

Eelco Visser



CS4200 | Compiler Construction | September 9, 2021

Disambiguation and Layout-Sensitive Syntax

Syntax Definition Summary

Derivations

- Generating sentences and trees from context-free grammars

Ambiguity

Declarative Disambiguation Rules

- Associativity and priority

Grammar Transformations

- Eliminating ambiguity by transformation

Layout-Sensitive Syntax

- Disambiguation using layout constraints

Structure

Syntax = Structure

```
module structure  
  
imports Common  
  
context-free start-symbols Exp
```

context-free syntax

Exp.Var = ID

Exp.Int = INT

Exp.Add = Exp "+" Exp

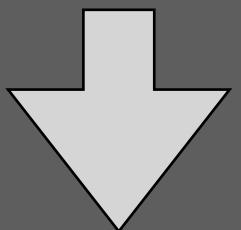
Exp.Fun = "function" "(" {ID ","}* ")" "{" Exp "}"

Exp.App = Exp "(" {Exp ","}* ")"

Exp.Let = "let" Bnd* "in" Exp "end"

Bnd.Bnd = ID "=" Exp

```
let  
  inc = function(x) { x + 1 }  
  in  
  inc(3)  
end
```



```
LetC  
  [ Bnd(  
    "inc"  
    , Fun(["x"], Add(Var("x"), Int("1")))  
  )  
  ]  
  , App(Var("inc"), [Int("3")])  
)
```

Token = Character

```
module structure  
  
imports Common  
  
context-free start-symbols Exp
```

context-free syntax

```
Exp.Var = ID  
  
Exp.Int = INT  
  
Exp.Add = Exp "+" Exp  
  
Exp.Fun = "function" "(" {ID ","}* ")" " {" Exp "}"  
  
Exp.App = Exp "(" {Exp ","}* ")"  
  
Exp.Let = "let" Bnd* "in" Exp "end"  
  
Bnd.Bnd = ID "=" Exp
```

```
let  
  inc = function(x) { x + 1 }  
in  
  inc(3)  
end
```

```
module Common
```

lexical syntax

```
ID   = [a-zA-Z] [a-zA-Z0-9]*  
  
INT = [\-\]? [0-9]+
```

Lexical Syntax = Context-Free Syntax
(But we don't care about structure of lexical syntax)

Literal = Non-Terminal

```
module structure  
  
imports Common  
  
context-free start-symbols Exp
```

context-free syntax

```
Exp.Var = ID  
  
Exp.Int = INT  
  
Exp.Add = Exp "+" Exp  
  
Exp.Fun = "function" "(" {ID ","}* ")" " {" Exp "}"  
  
Exp.App = Exp "(" {Exp ","}* ")"  
  
Exp.Let = "let" Bnd* "in" Exp "end"  
  
Bnd.Bnd = ID "=" Exp
```

```
let  
  inc = function(x) { x + 1 }  
in  
  inc(3)  
end
```

syntax

"+"	= [\u0043]
"function"	= [\u00102] [\u00117] [\u00110] [\u00199] [\u00116] [\u00105] [\u00111] [\u00110]
"{"	= [\u00123]
= [\u00125]	
"("	= [\u00140]
= [\u00144]	
= [\u00141]	
"let"	= [\u00108] [\u00101] [\u00116]
"in"	= [\u00105] [\u00110]
"end"	= [\u00101] [\u00110] [\u00100]
"="	= [\u00161]

Layout = Whitespace & Comments

```
module Common

lexical syntax

LAYOUT      = [\t\n\r]
LAYOUT      = /* InsideComment* */
InsideComment = ~[*]
InsideComment = CommentChar
CommentChar  = [*]

LAYOUT      = // ~[\n\r]* NewLineEOF
NewLineEOF   = [\n\r]
NewLineEOF   = EOF
```

```
let
  inc = function(x) { x + 1 }
in
// function application
inc /* function position */ (
  3 // argument list
)
end
```

Layout = (Almost) Everywhere

```
module Common

lexical syntax

LAYOUT      = [\t\n\r]
LAYOUT      = /* InsideComment* */
InsideComment = ~[*]
InsideComment = CommentChar
CommentChar  = [*]

LAYOUT      = // ~[\n\r]* NewLineEOF
NewLineEOF   = [\n\r]
NewLineEOF   = EOF
```

```
let
  inc = function(x) { x + 1 }
in
// function application
inc /* function position */ (
  3 // argument list
)
end
```

```
Exp.App = Exp "(" {Exp ","}* ")"
```

```
Exp-CF.App = Exp-CF LAYOUT?-CF "(" LAYOUT?-CF {Exp ","}* -CF LAYOUT?-CF ")"
```

Extension

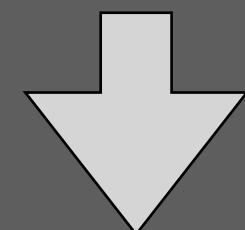
Language Composition ⇒ Grammar Composition

```
module extension
imports functional query
context-free start-symbols Exp
context-free syntax
Exp = Query
Cond = Exp
```

```
module functional
imports Common
context-free syntax
Exp = <(<Exp>)> {bracket}
...
```

```
module query
imports Common
context-free syntax
Query.Query = <
  select <QID*> from <QID*> where <Cond>
>
Cond.And = <<Cond> and <Cond>> {left}
Cond.Eq = <<Cond> == <Cond>> {non-assoc}
```

```
let
  select = 1
  fs = select f from A where test f = select
in
  print fs
```



```
Let(
  [ Bnd("select", Int("1"))
  , Bnd(
    "fs"
  , Query(
    ["f"]
    , ["A"]
    , Eq(App(Var("test"), Var("f")), Var("select"))
    )
  )
  ]
  , App(Var("print"), Var("fs"))
)
```

Parsing = Formatting⁻¹

Parsing = Formatting⁻¹

context-free syntax

Exp.Var = <<ID>>

Exp.Int = <<INT>>

Exp.Add = <<Exp> + <Exp>>

Exp.Fun = <
function(<{ID ", "}*>){
 <Exp>
}
>

Exp.App = <<Exp>(<{Exp ", "}*>)>

Exp.Let = <
let
 <{Bnd "\n"}*>
 in
 <Exp>
end
>

Bnd.Bnd = <<ID> = <Exp>>

```
let
  inc = function(x) { x + 1 }
in
inc(3)
end
```



```
Let(
  [ Bnd(
    "inc"
    , Fun(["x"], Add(Var("x"), Int("1")))
  )
  ]
, App(Var("inc"), [Int("3")])
)
```



```
let
  inc = function(x){
    x + 1
  }
in
inc(3)
end
```

Completion = Rewrite(Incomplete Structure)

```
class A {  
  
    public int m() {  
        int x;  
        x = $Exp;  
        return+$Add  
            +$Sub  
            +$Mul  
            +$Lt  
            +$VarRef
```

```
class A {  
  
    public int m() {  
        int x;  
        x = $Exp + $Exp;  
        retu+$Add  
            +$Sub  
            +$Mul  
            +$Lt  
            +$VarRef
```

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + $Exp;  
        return x;+$Add  
            +$Sub  
            +$Mul  
            +$Lt  
            +$VarRef
```

```
class A {  
  
    public int m() {  
        int x;  
        x = 21 + 21;  
        return x;  
    }
```

Context-Free Grammars

Terminals

- Basic symbols from which strings are formed

Nonterminals

- Syntactic variables that denote sets of strings

Start Symbol

- Denotes the nonterminal that generates strings of the languages

Productions

- $A = X \dots X$
- Head/left side (A) is a nonterminal
- Body/right side ($X \dots X$) zero or more terminals and nonterminals

Example Context-Free Grammar

```
grammar
  start S
  non-terminals E T F
  terminals "+" "*" "(" ")" ID
  productions
    S = E
    E = E "+" T
    E = T
    T = T "*" F
    T = F
    F = "(" E ")"
    F = ID
```

Abbreviated Grammar

```
grammar
start S
non-terminals E T F
terminals "+" "*" "(" ")" ID
productions
S = E
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```

```
grammar
productions
S = E
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```

Nonterminals, terminals can be derived from productions

First production defines start symbol

Notation

A, B, C: non-terminals

l: terminals

a, b, c: strings of non-terminals and terminals
(alpha, beta, gamma in math)

w, v: strings of terminal symbols

Meta: Syntax of Grammars

context-free syntax // grammars

```
Grammar.Grammar = <  
    grammar  
    <Start?>  
    <Sorts?>  
    <Terminals?>  
    <Productions>  
>
```

Meta: Syntax of Grammars

context-free syntax // grammars

```
Grammar.Grammar = <  
    grammar  
    <Start?>  
    <Sorts?>  
    <Terminals?>  
    <Productions>  
>
```

context-free syntax

```
Start.Start = <  
    start <ID>  
>
```

```
Sorts.Sorts = <  
    sorts <ID*>  
>
```

```
Sorts.NonTerminals = <  
    non-terminals <ID*>  
>
```

```
Terminals.Terminals = <  
    terminals <Symbol*>  
>
```

```
Productions.Productions = <  
    productions  
    <Production*>  
>
```

Meta: Syntax of Grammars

context-free syntax // grammars

```
Grammar.Grammar = <
  grammar
    <Start?>
    <Sorts?>
    <Terminals?>
    <Productions>
  >
```

context-free syntax

```
Production.Prod = <
  <Symbol><Constructor?> = <Symbol*>
>
```

```
Symbol.NT = <<ID>>
Symbol.T = <<STRING>>
Symbol.L = <<LCID>>
```

```
Constructor.Con = <.<ID>>
```

context-free syntax

```
Start.Start = <
  start <ID>
>
```

```
Sorts.Sorts = <
  sorts <ID*>
>
```

```
Sorts.NonTerminals = <
  non-terminals <ID*>
>
```

```
Terminals.Terminals = <
  terminals <Symbol*>
>
```

```
Productions.Productions = <
  productions
  <Production*>
>
```

Derivations: Generating Sentences from Symbols

Derivations

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
// derivation step: replace symbol by rhs of production  
// E = E "+" E  
// replace E by E "+" E  
//  
// derivation:  
// repeatedly apply derivations
```

Derivations

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
// derivation step: replace symbol by rhs of production  
// E = E "+" E  
// replace E by E "+" E  
//  
// derivation:  
// repeatedly apply derivations
```

```
derivation
```

Derivations

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
// derivation step: replace symbol by rhs of production  
// E = E "+" E  
// replace E by E "+" E  
//  
// derivation:  
// repeatedly apply derivations
```

```
derivation  
E
```

Derivations

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
// derivation step: replace symbol by rhs of production  
// E = E "+" E  
// replace E by E "+" E  
//  
// derivation:  
// repeatedly apply derivations
```

```
derivation  
E  
⇒ "-" E
```

Derivations

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
// derivation step: replace symbol by rhs of production  
// E = E "+" E  
// replace E by E "+" E  
//  
// derivation:  
// repeatedly apply derivations
```

```
derivation  
E  
⇒ "-" E  
⇒ "-" "(" E ")"
```

Derivations

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
// derivation step: replace symbol by rhs of production  
// E = E "+" E  
// replace E by E "+" E  
//  
// derivation:  
// repeatedly apply derivations
```

```
derivation  
E  
⇒ "-" E  
⇒ "-" "(" E ")"  
⇒ "-" "(" ID ")"
```

Derivations

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
// derivation step: replace symbol by rhs of production  
// E = E "+" E  
// replace E by E "+" E  
//  
// derivation:  
// repeatedly apply derivations
```

```
derivation  
E  
⇒ "-" E  
⇒ "-" "(" E ")"  
⇒ "-" "(" ID ")"
```

```
derivation // derives in zero or more steps  
E ⇒* "-" "(" ID "+" ID ")"
```

Meta: Syntax of Derivations

context-free syntax // derivations

```
Derivation.Derivation = <  
    derivation  
        <Symbol> <Step*>  
>
```

```
Step.Step      = [⇒ [Symbol*]]  
Step.Steps    = [⇒* [Symbol*]]  
Step.Steps1   = [⇒+ [Symbol*]]
```

Left-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

Left-most derivation: Expand left-most non-terminal at each step

Left-Most Derivation

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
derivation // left-most derivation
```

Left-most derivation: Expand left-most non-terminal at each step

Left-Most Derivation

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
derivation // left-most derivation
```

```
E
```

Left-most derivation: Expand left-most non-terminal at each step

Left-Most Derivation

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
derivation // left-most derivation  
E  
⇒ "-" E
```

Left-most derivation: Expand left-most non-terminal at each step

Left-Most Derivation

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
derivation // left-most derivation  
E  
⇒ "-" E  
⇒ "-" "(" E ")"
```

Left-most derivation: Expand left-most non-terminal at each step

Left-Most Derivation

```
grammar  
productions  
E = E "+" E  
E = E "*" E  
E = "-" E  
E = "(" E ")"  
E = ID
```

```
derivation // left-most derivation  
E  
⇒ "-" E  
⇒ "-" "(" E ")"  
⇒ "-" "(" E "+" E ")"
```

Left-most derivation: Expand left-most non-terminal at each step

Left-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
```

Left-most derivation: Expand left-most non-terminal at each step

Left-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

Left-most derivation: Expand left-most non-terminal at each step

Right-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

Right-most derivation: Expand right-most non-terminal at each step

Right-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

derivation // right-most derivation

Right-most derivation: Expand right-most non-terminal at each step

Right-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

derivation // right-most derivation

```
E
```

Right-most derivation: Expand right-most non-terminal at each step

Right-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

derivation // right-most derivation

```
E
⇒ "-" E
```

Right-most derivation: Expand right-most non-terminal at each step

Right-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

derivation // right-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
```

Right-most derivation: Expand right-most non-terminal at each step

Right-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

derivation // right-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
```

Right-most derivation: Expand right-most non-terminal at each step

Right-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

derivation // right-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" E "+" ID ")"
```

Right-most derivation: Expand right-most non-terminal at each step

Right-Most Derivation

grammar
productions

```
E = E "+" E
E = E "*" E
E = "-" E
E = "(" E ")"
E = ID
```

derivation // left-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" ID "+" E ")"
⇒ "-" "(" ID "+" ID ")"
```

derivation // right-most derivation

```
E
⇒ "-" E
⇒ "-" "(" E ")"
⇒ "-" "(" E "+" E ")"
⇒ "-" "(" E "+" ID ")"
⇒ "-" "(" ID "+" ID ")"
```

Right-most derivation: Expand right-most non-terminal at each step

Meta: Tree Derivations

```
context-free syntax // tree derivations
```

```
Derivation.TreeDerivation = <  
    tree derivation  
    <Symbol> <PStep*>  
>
```

```
PStep.Step      = [⇒ [PT*]]  
PStep.Steps    = [⇒* [PT*]]  
PStep.Steps1   = [⇒+ [PT*]]
```

```
PT.App = <<Symbol>[<PT*>]>  
PT.Str = <<STRING>>  
PT.Sym = <<Symbol>>
```

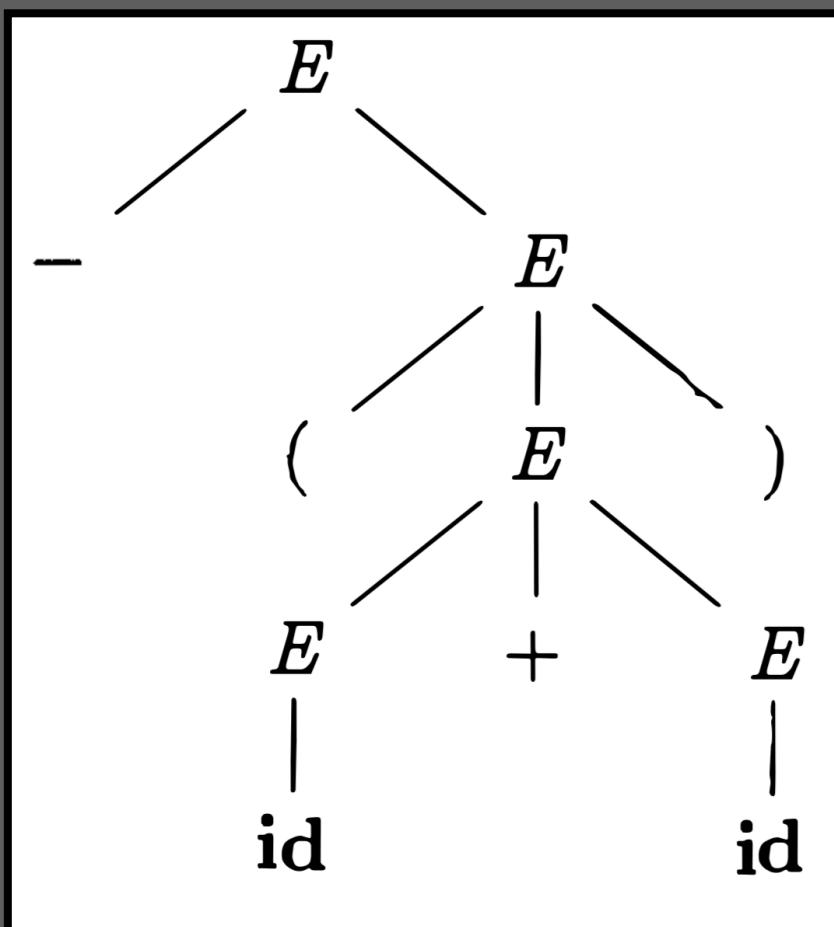
Left-Most Tree Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



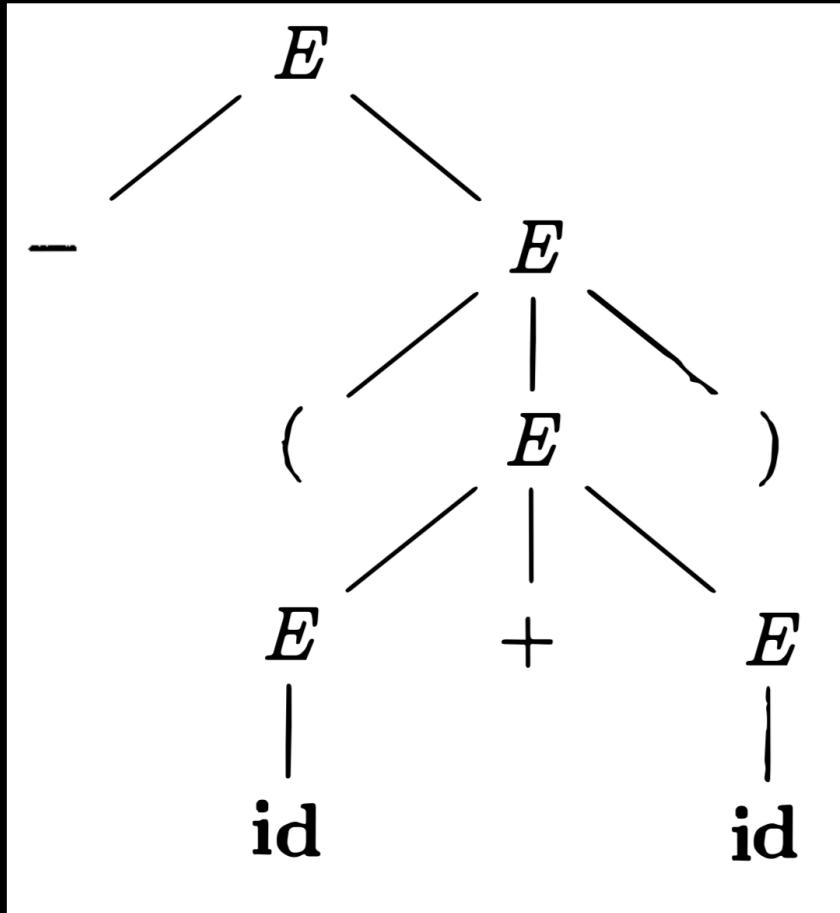
Left-Most Tree Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



tree derivation // left-most

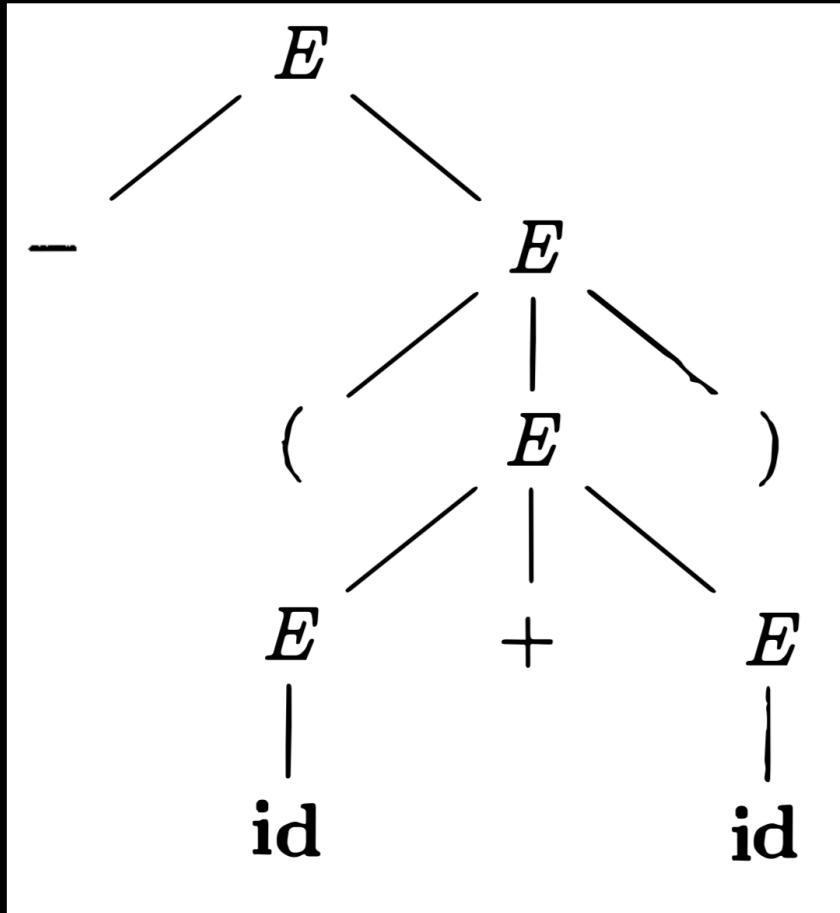
Left-Most Tree Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



tree derivation // left-most

E

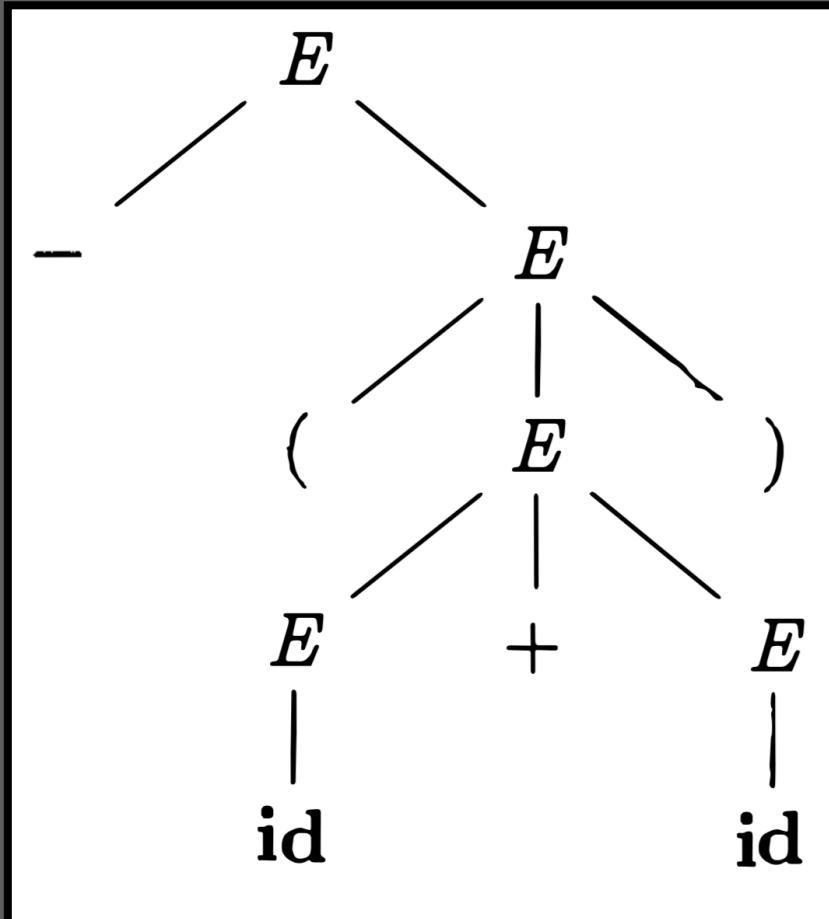
Left-Most Tree Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



tree derivation // left-most

E
 $\Rightarrow E["-" E]$

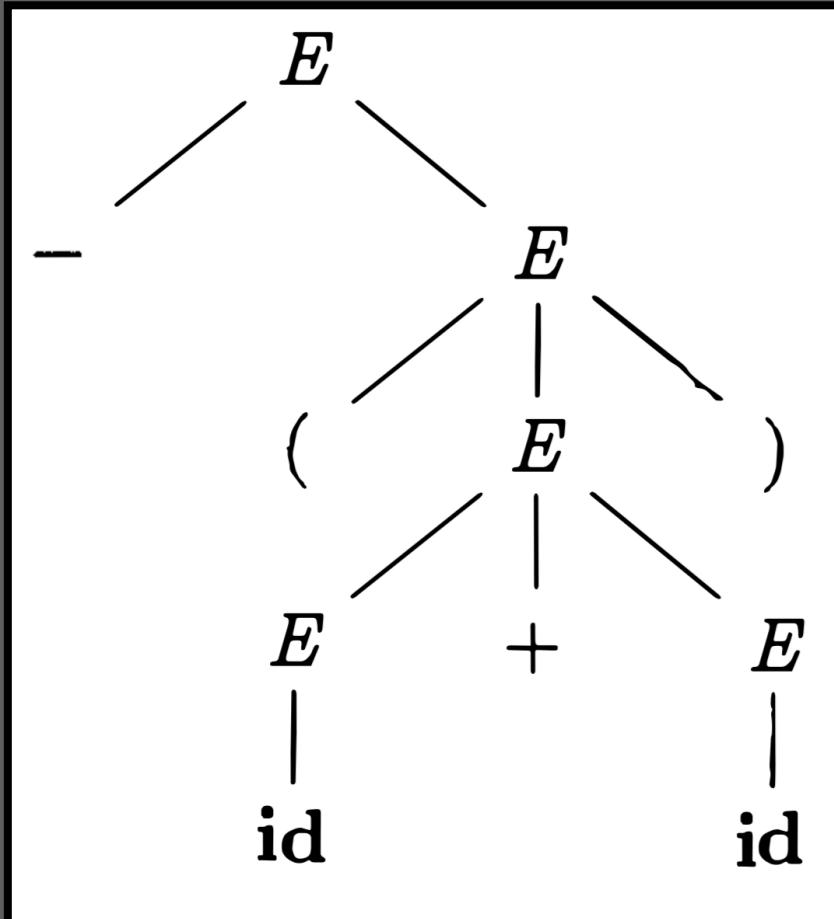
Left-Most Tree Derivation

grammar
productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = " (" \ E \ ") "$
 $E \cdot V = ID$

derivation // left-most derivation

E
 $\Rightarrow " - " \ E$
 $\Rightarrow " - " \ " (" \ E \ ") "$
 $\Rightarrow " - " \ " (" \ E \ " + " \ E \ ") "$
 $\Rightarrow " - " \ " (" \ ID \ " + " \ E \ ") "$
 $\Rightarrow " - " \ " (" \ ID \ " + " \ ID \ ") "$



tree derivation // left-most

E
 $\Rightarrow E[" - " \ E]$
 $\Rightarrow E[" - " \ E[" (" \ E \ ") "]]$

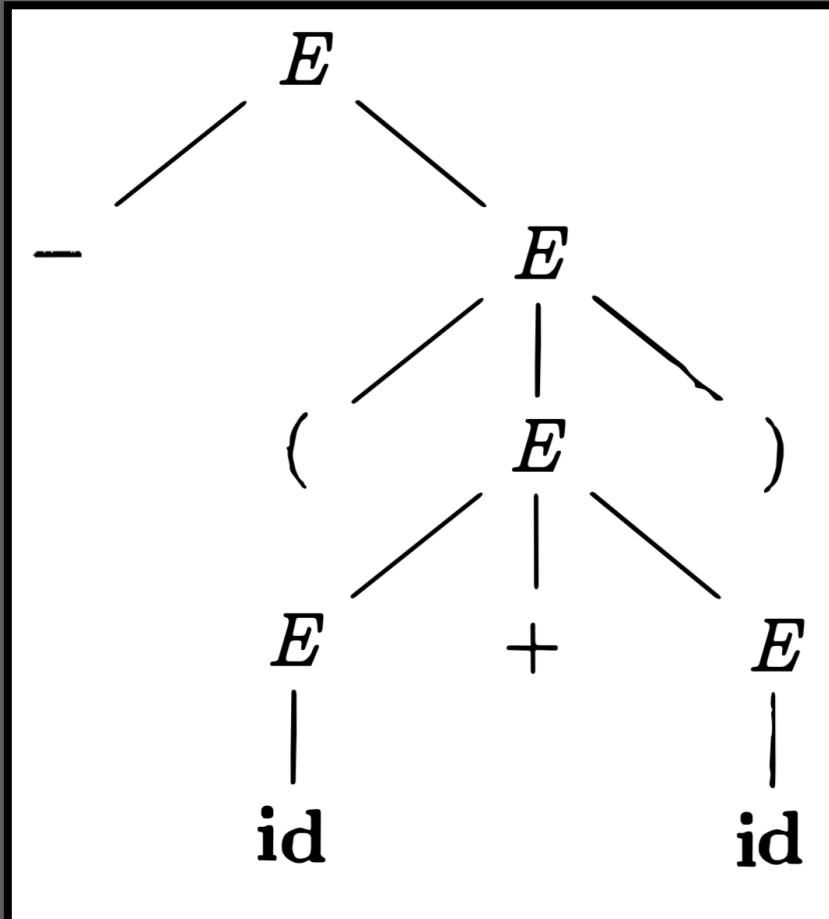
Left-Most Tree Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



tree derivation // left-most

E
 $\Rightarrow E["-" E]$
 $\Rightarrow E["-" E["(" E ")"]]$
 $\Rightarrow E["-" E["(" E[E "+" E] ")"]]$

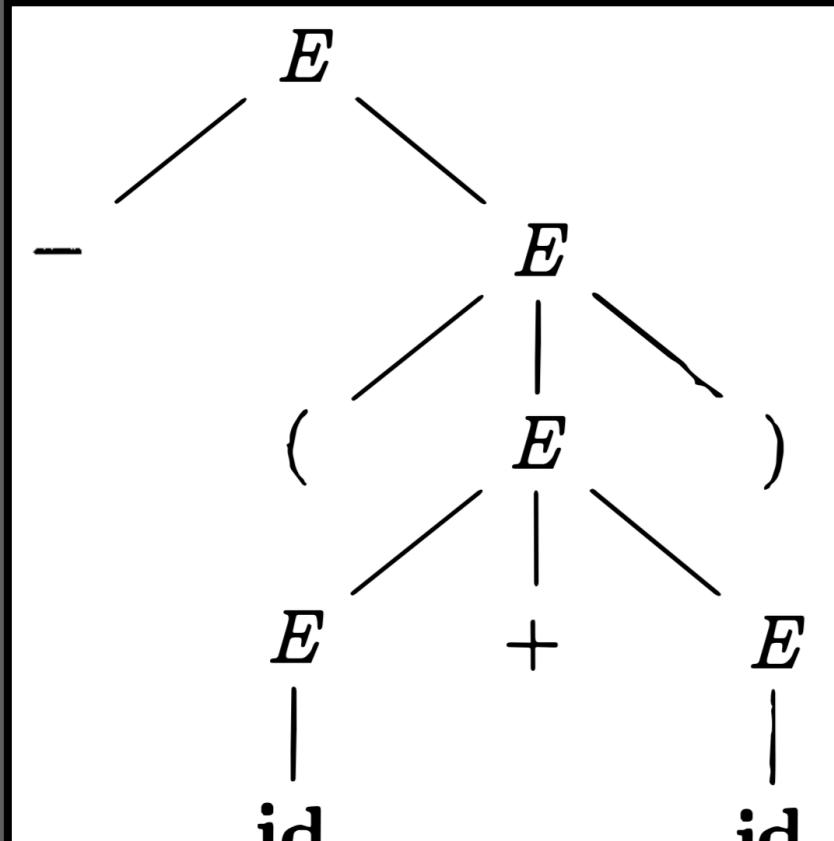
Left-Most Tree Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



tree derivation // left-most

E
 $\Rightarrow E["-" E]$
 $\Rightarrow E["-" E["(" E ")"]]$
 $\Rightarrow E["-" E["(" E[E "+" E] ")"]]$
 $\Rightarrow E["-" E["(" E[E[ID] "+" E] ")"]]$

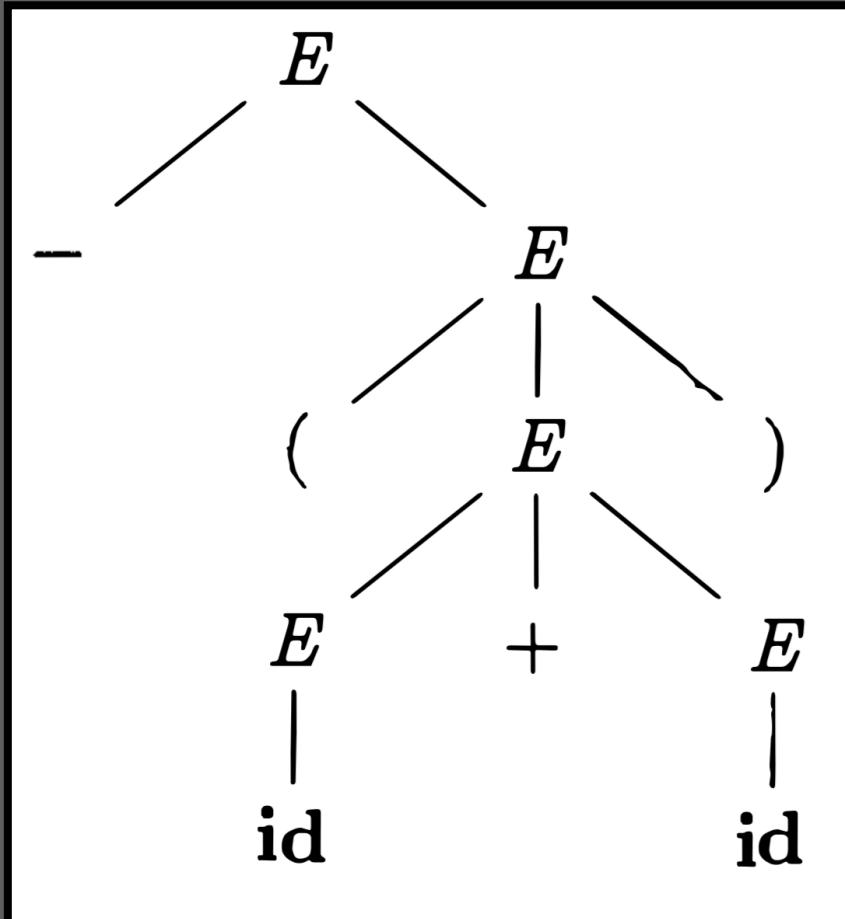
Left-Most Tree Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

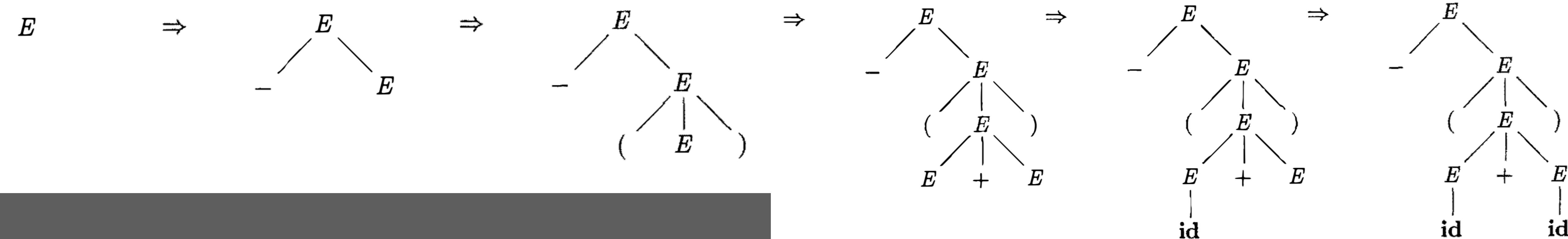
E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



tree derivation // left-most

E
 $\Rightarrow E["-" E]$
 $\Rightarrow E["-" E["(" E ")"]]$
 $\Rightarrow E["-" E["(" E[E "+" E] ")"]]$
 $\Rightarrow E["-" E["(" E[E[ID] "+" E] ")"]]$
 $\Rightarrow E["-" E["(" E[E[ID] "+" E[ID]] ")"]]$

Left-Most Tree Derivation



tree derivation // left-most

```
E
⇒ E["- E"]
⇒ E["- E[(" E ")"]"]
⇒ E["- E[(" E[E "+" E] ")"]"]
⇒ E["- E[(" E[E[ID] "+" E] ")"]"]
⇒ E["- E[(" E[E[ID]] "+" E[ID]) ")"]"]
```

Meta: Term Derivations

```
context-free syntax // term derivations
```

```
Derivation.TermDerivation = <  
    term derivation  
    <Symbol> <TStep*>  
>
```

```
TStep.Step      = [⇒ [Term*]]  
TStep.Steps    = [⇒* [Term*]]  
TStep.Steps1   = [⇒+ [Term*]]
```

```
Term.App = <<ID>(<{Term ",,"}*>)>  
Term.Str = <<STRING>>  
Term.Sym = <<Symbol>>
```

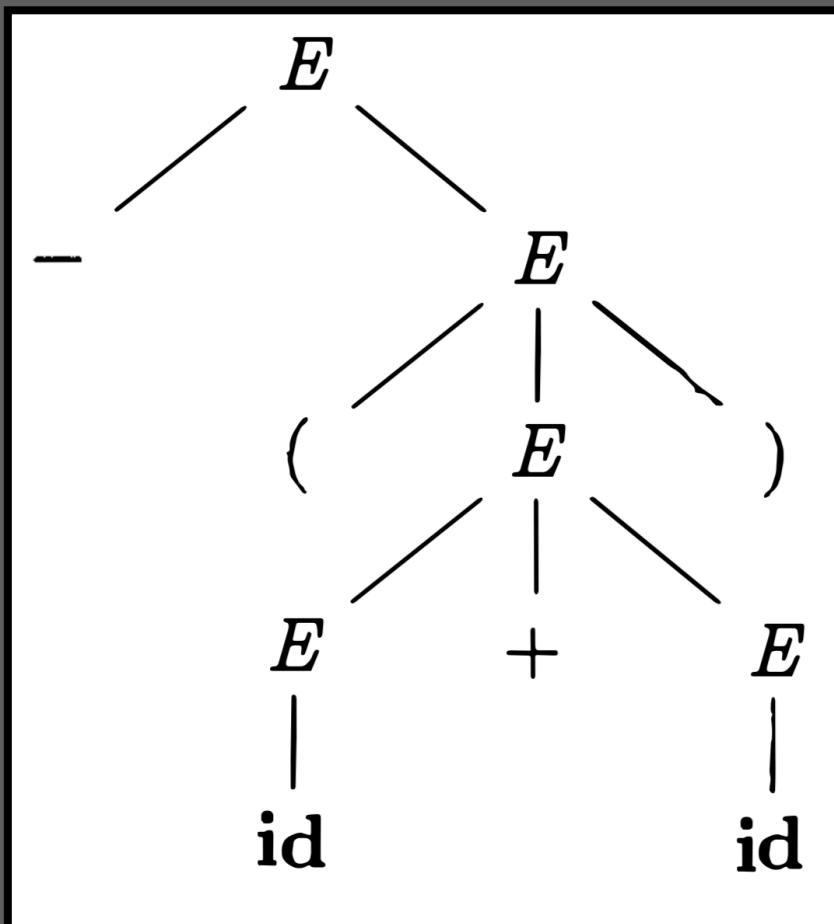
Left-Most Term Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



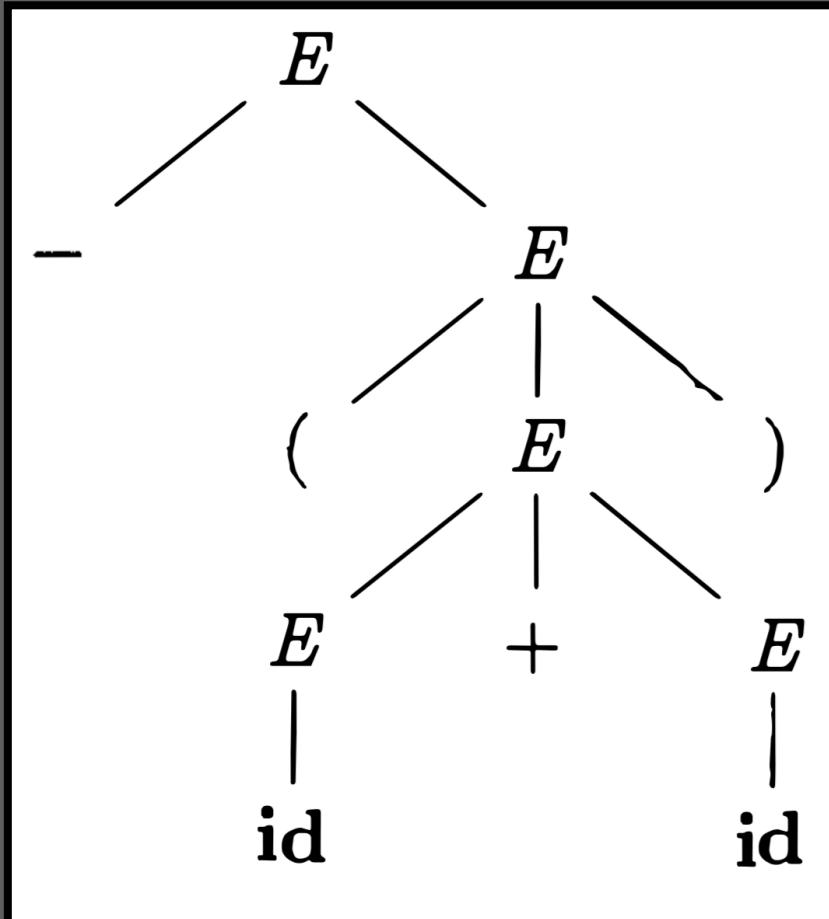
Left-Most Term Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



term derivation // left-most

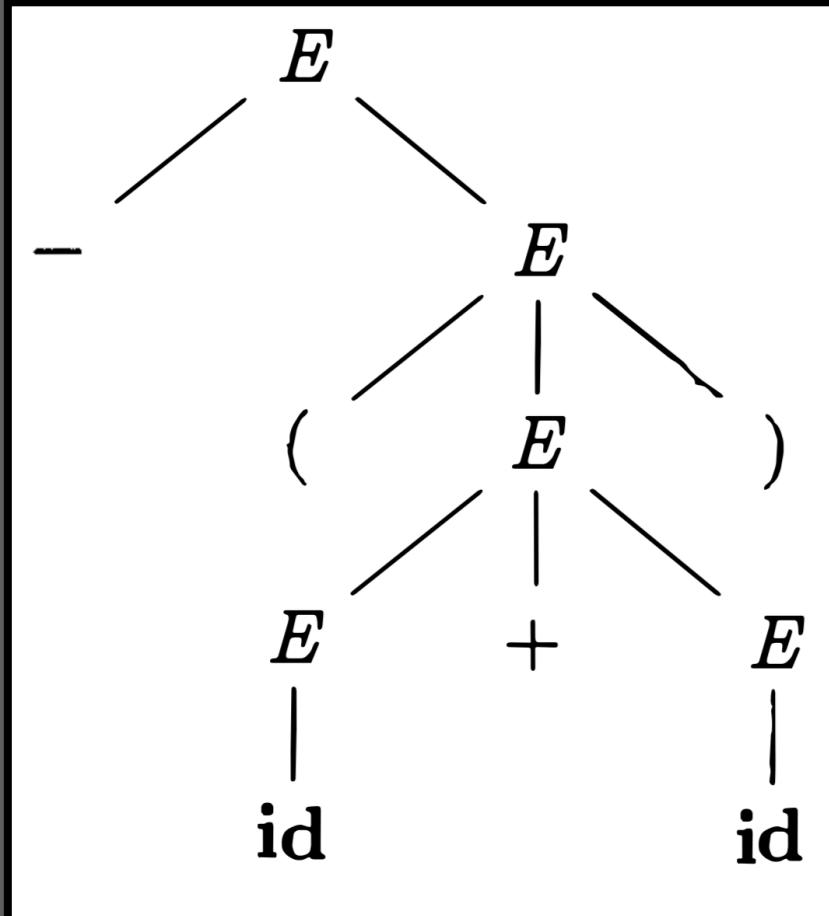
Left-Most Term Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



term derivation // left-most

E

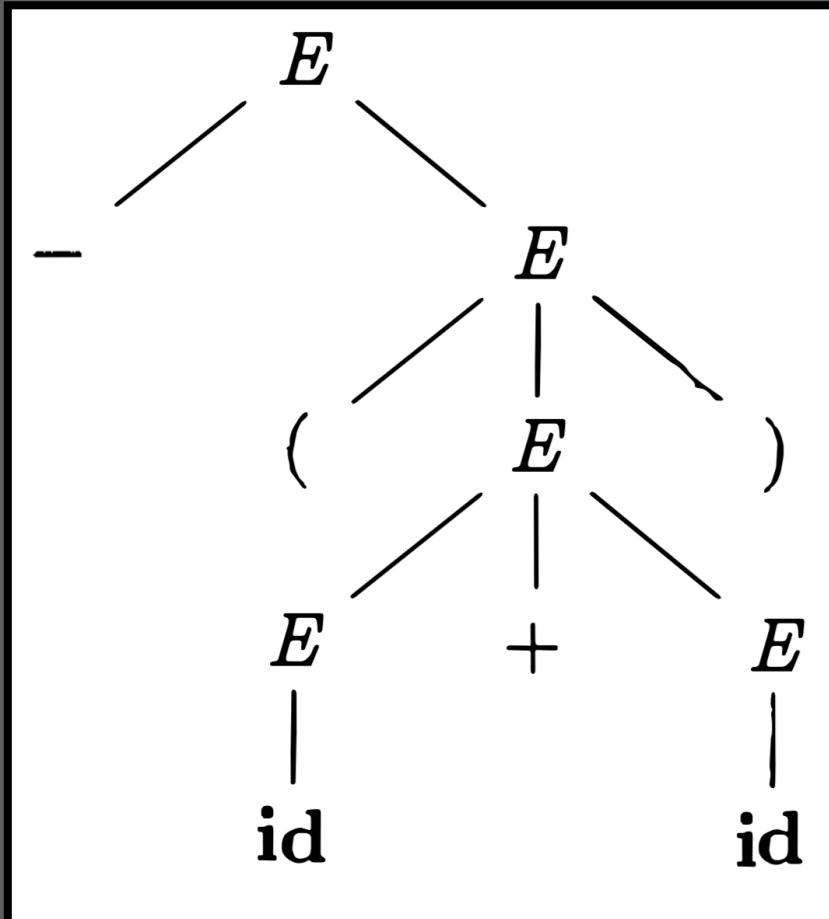
Left-Most Term Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



term derivation // left-most

E
 $\Rightarrow N(E)$

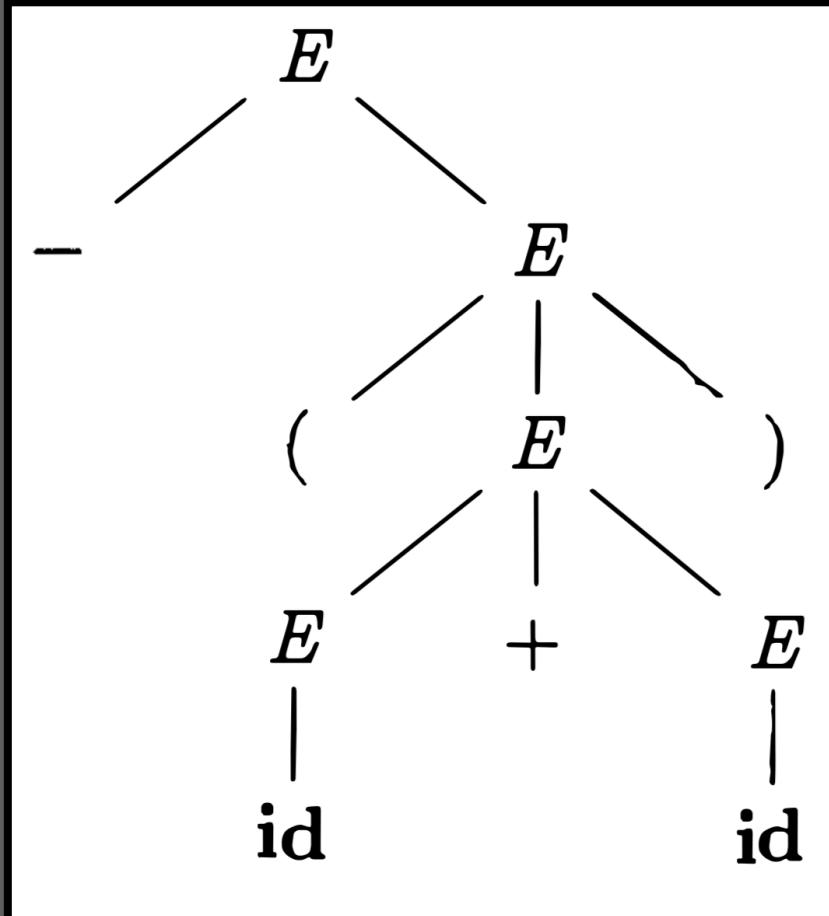
Left-Most Term Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



term derivation // left-most

E
 $\Rightarrow N(E)$
 $\Rightarrow N(P(E))$

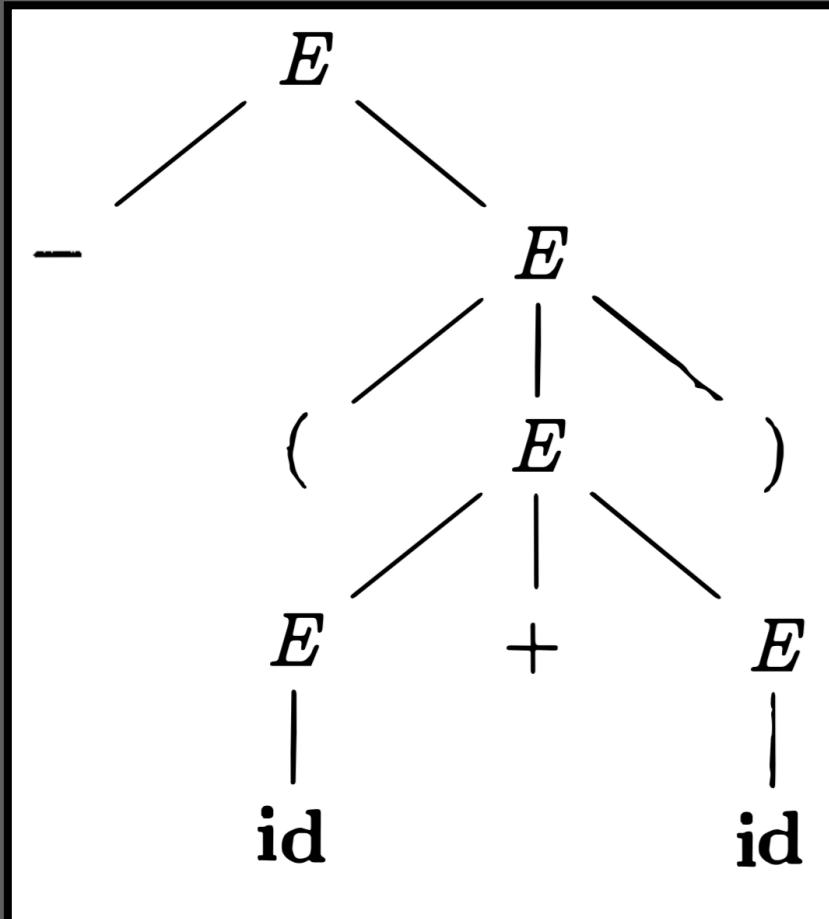
Left-Most Term Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



term derivation // left-most

E
 $\Rightarrow N(E)$
 $\Rightarrow N(P(E))$
 $\Rightarrow N(P(A(E, E)))$

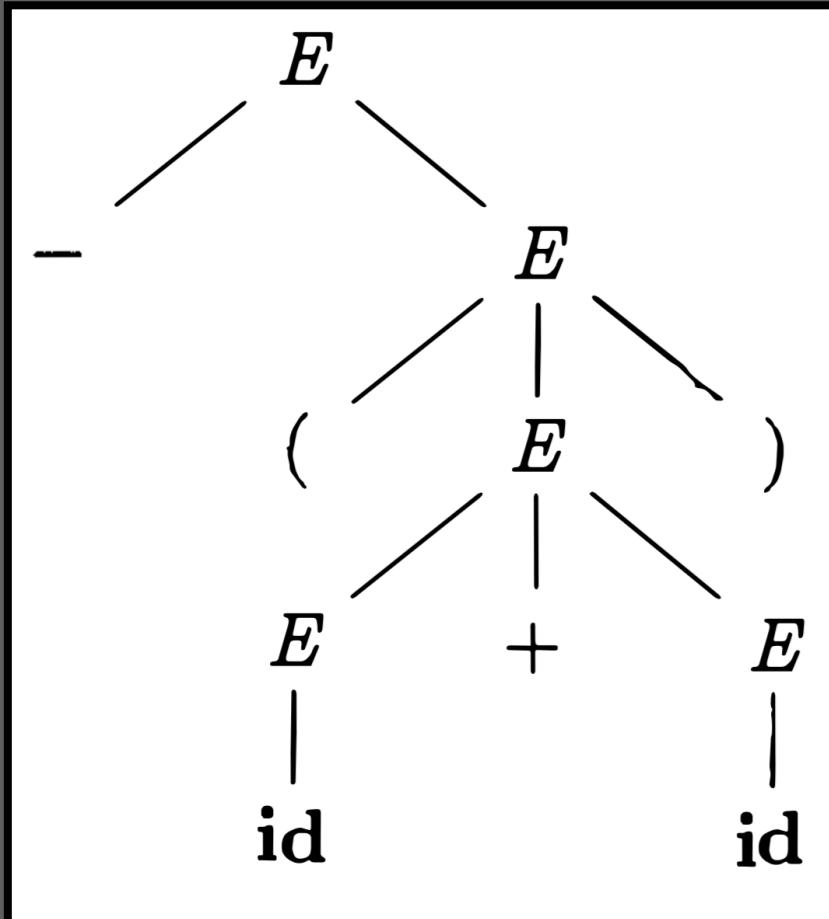
Left-Most Term Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



term derivation // left-most

E
 $\Rightarrow N(E)$
 $\Rightarrow N(P(E))$
 $\Rightarrow N(P(A(E, E)))$
 $\Rightarrow N(P(A(V(ID)), E))$

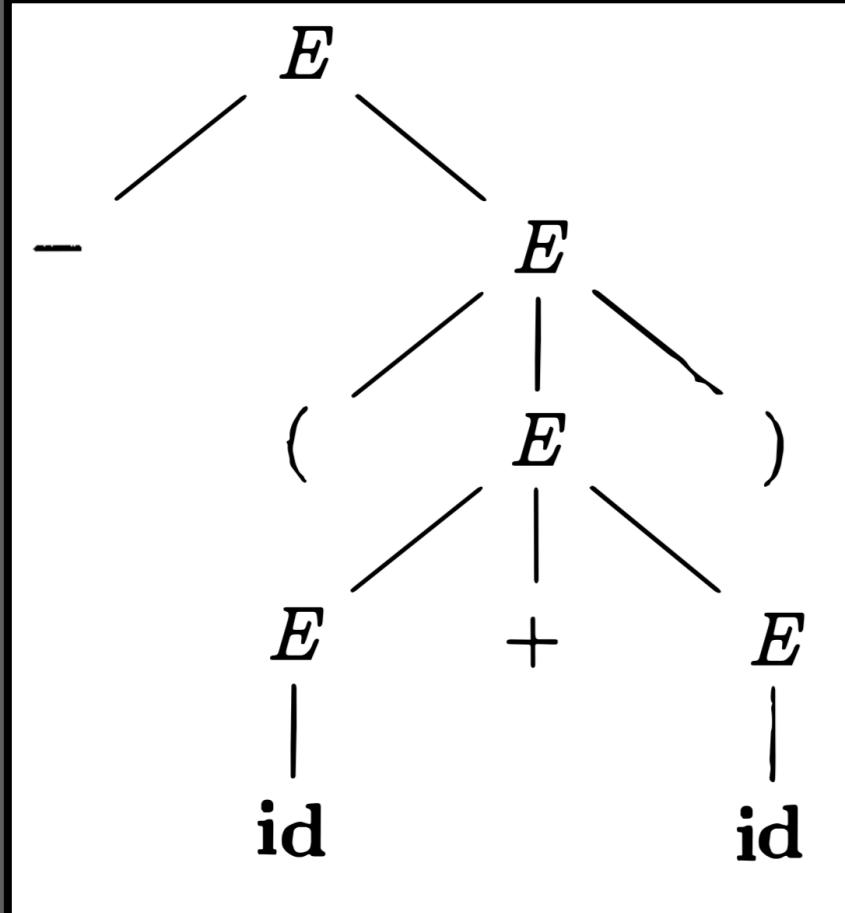
Left-Most Term Derivation

grammar
productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation // left-most derivation

E
 $\Rightarrow "-" E$
 $\Rightarrow "-" "(" E ")"$
 $\Rightarrow "-" "(" E "+" E ")"$
 $\Rightarrow "-" "(" ID "+" E ")"$
 $\Rightarrow "-" "(" ID "+" ID ")"$



term derivation // left-most

E
 $\Rightarrow N(E)$
 $\Rightarrow N(P(E))$
 $\Rightarrow N(P(A(E, E)))$
 $\Rightarrow N(P(A(V(ID), E)))$
 $\Rightarrow N(P(A(V(ID), V(ID))))$

Parse Trees Represent Derivations

```
List<String> YIELD(T : Tree) {  
    T match {  
        [A = Ts] => YIELDS(Ts);  
        Str => [Str];  
    };  
}  
  
List<String> YIELDS(Ts : List<Tree>) {  
    Ts match {  
        [] => "";  
        [T | Ts] => YIELD(T) + YIELDS(Ts);  
    };  
}
```

$$S \xrightarrow{*} PT$$

iff

$$S \xrightarrow{*} \text{YIELD}(PT)$$

Language Defined by a Grammar

$$L(G) = \{ w \mid S \xrightarrow{*} w \}$$

Language: sentences

$$T(G) = \{ T \mid S \xrightarrow{*} T \}$$

Language: trees

$$L(G) = \text{YIELD}(T(G))$$

Ambiguity

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

```
E.A = E "+" E  
E.T = E "*" E  
E.N = "-" E  
E.P = "(" E ")"  
E.V = ID
```

derivation

$$E \xrightarrow{*} ID\ "+" ID\ "*" ID$$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

E

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

E
 $\Rightarrow E \ " + " \ E$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

E
 $\Rightarrow E \ " + " \ E$
 $\Rightarrow ID \ " + " \ E$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

E
 $\Rightarrow E \ " + " \ E$
 $\Rightarrow ID \ " + " \ E$
 $\Rightarrow ID \ " + " \ E \ " * " \ E$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

E
 $\Rightarrow E \ " + " \ E$
 $\Rightarrow ID \ " + " \ E$
 $\Rightarrow ID \ " + " \ E \ " * " \ E$
 $\Rightarrow ID \ " + " \ ID \ " * " \ E$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation

$E \Rightarrow^* ID "+" ID "*" ID$

derivation

E
 $\Rightarrow E "+" E$
 $\Rightarrow ID "+" E$
 $\Rightarrow ID "+" E "*" E$
 $\Rightarrow ID "+" ID "*" E$
 $\Rightarrow ID "+" ID "*" ID$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$

$E \cdot T = E \ " * " \ E$

$E \cdot N = " - " \ E$

$E \cdot P = "(" \ E \ ")"$

$E \cdot V = ID$

derivation

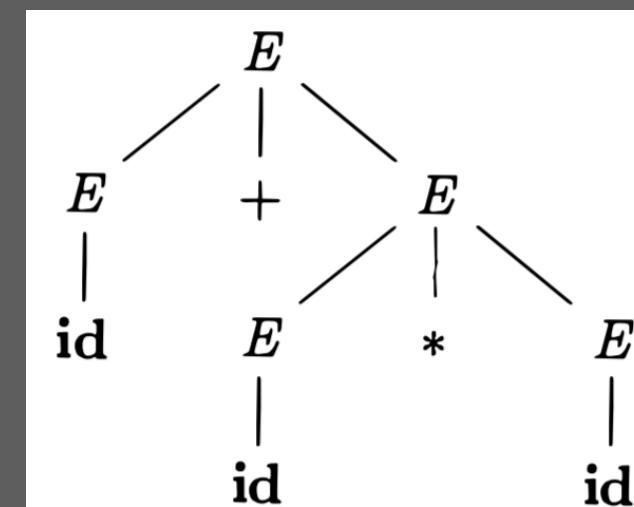
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$$\begin{aligned} E &\Rightarrow E \ " + " \ E \\ &\Rightarrow ID \ " + " \ E \\ &\Rightarrow ID \ " + " \ E \ " * " \ E \\ &\Rightarrow ID \ " + " \ ID \ " * " \ E \\ &\Rightarrow ID \ " + " \ ID \ " * " \ ID \end{aligned}$$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$



Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$

$E \cdot T = E \ " * " \ E$

$E \cdot N = " - " \ E$

$E \cdot P = "(" \ E \ ")"$

$E \cdot V = ID$

derivation

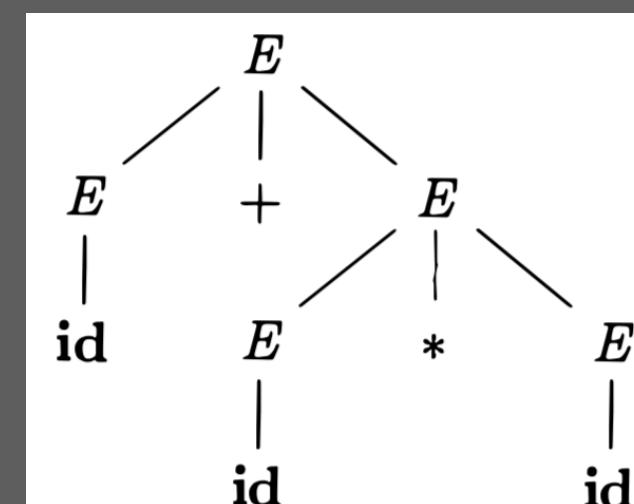
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$$\begin{aligned} E &\Rightarrow E \ " + " \ E \\ &\Rightarrow ID \ " + " \ E \\ &\Rightarrow ID \ " + " \ E \ " * " \ E \\ &\Rightarrow ID \ " + " \ ID \ " * " \ E \\ &\Rightarrow ID \ " + " \ ID \ " * " \ ID \end{aligned}$$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$



derivation

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$

$E \cdot T = E \ " * " \ E$

$E \cdot N = " - " \ E$

$E \cdot P = "(" \ E \ ")"$

$E \cdot V = ID$

derivation

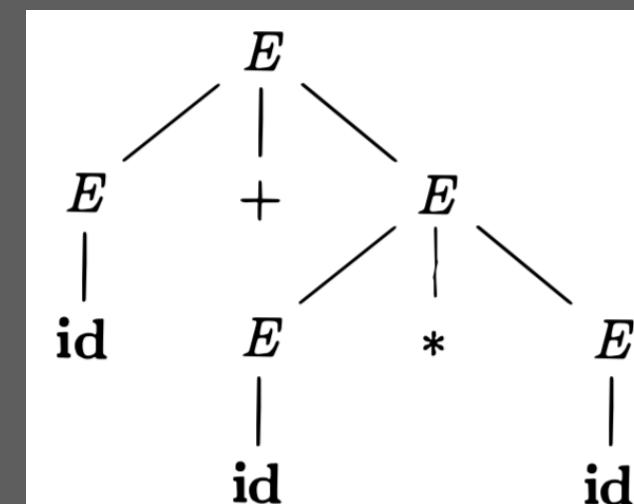
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$$\begin{aligned} E &\Rightarrow E \ " + " \ E \\ &\Rightarrow ID \ " + " \ E \\ &\Rightarrow ID \ " + " \ E \ " * " \ E \\ &\Rightarrow ID \ " + " \ ID \ " * " \ E \\ &\Rightarrow ID \ " + " \ ID \ " * " \ ID \end{aligned}$$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$



derivation

E

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

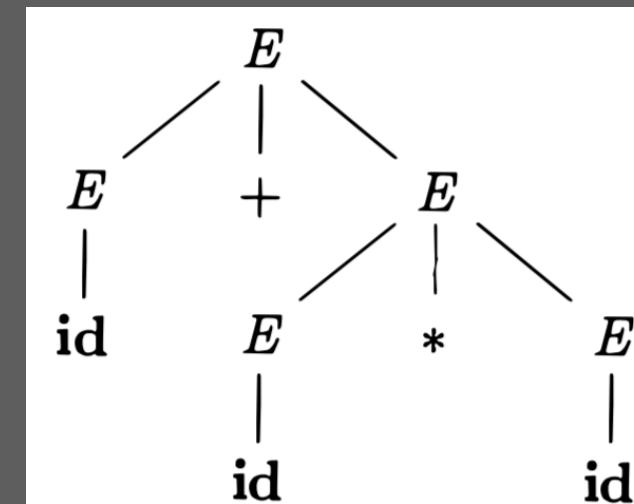
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$E \xrightarrow{*} E \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$



derivation

$E \xrightarrow{*} E \ " * " \ E$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

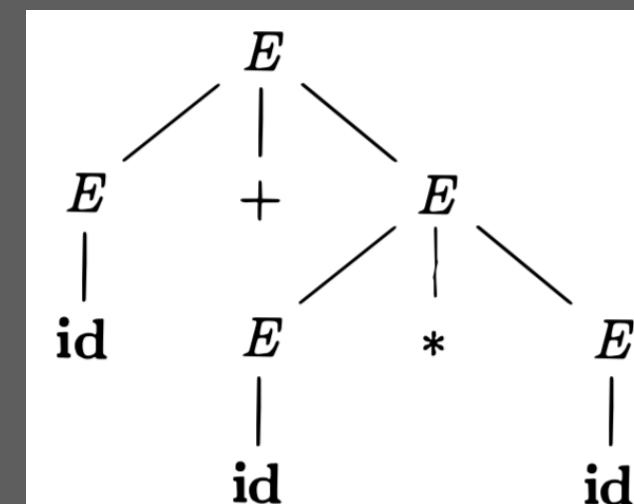
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$E \xrightarrow{*} E \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$



derivation

$E \xrightarrow{*} E \ " * " \ E$
 $\xrightarrow{*} E \ " + " \ E \ " * " \ E$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

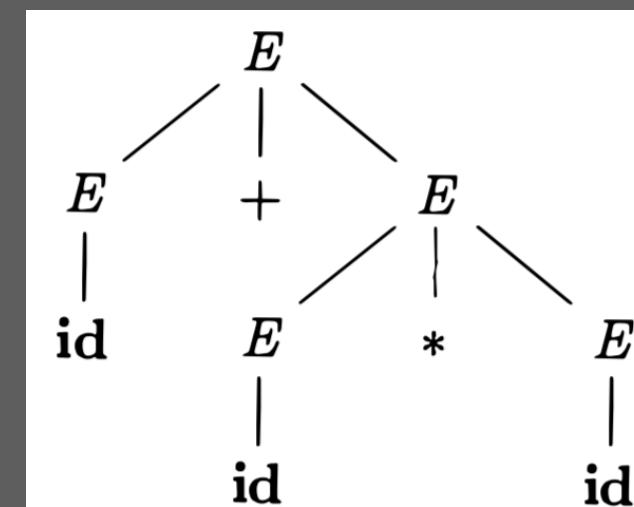
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$E \xrightarrow{*} E \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$



derivation

$E \xrightarrow{*} E \ " * " \ E$
 $\xrightarrow{*} E \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

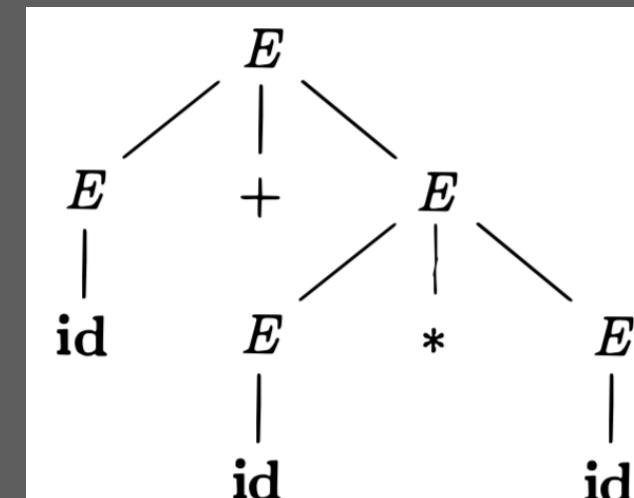
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$E \xrightarrow{*} E \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$



derivation

$E \xrightarrow{*} E \ " * " \ E$
 $\xrightarrow{*} E \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

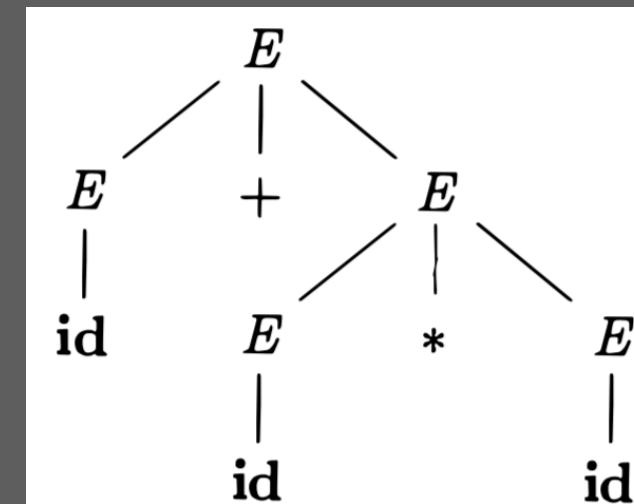
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$E \xrightarrow{*} E \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$



derivation

$E \xrightarrow{*} E \ " * " \ E$
 $\xrightarrow{*} E \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Multiple Parse Trees

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

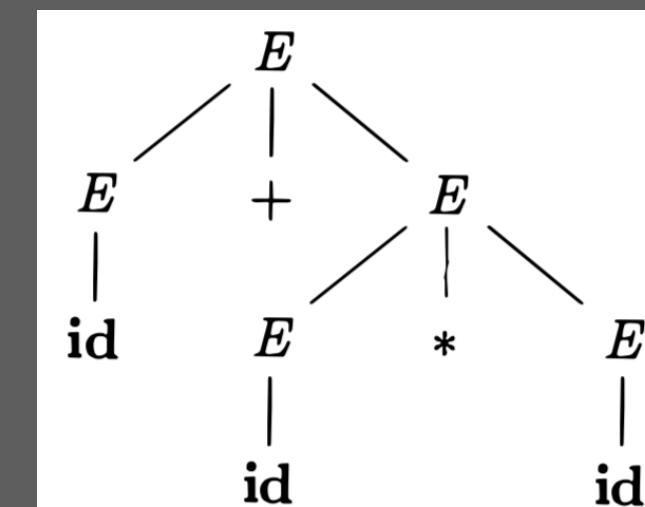
$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

$E \xrightarrow{*} E \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

tree derivation

$E \xrightarrow{*} E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]$

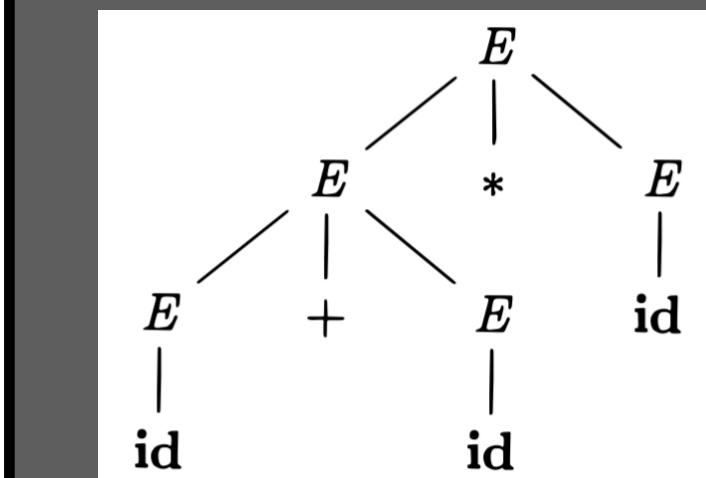


derivation

$E \xrightarrow{*} E \ " * " \ E$
 $\xrightarrow{*} E \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ E \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ E$
 $\xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

tree derivation

$E \xrightarrow{*} E[E[E[ID]] \ " + " \ E[E[ID]] \ " * " \ E[E[ID]]]$



Ambiguous grammar: produces >1 parse tree for a sentence

Ambiguity: Deriving Abstract Syntax Terms

grammar

productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation

$E \Rightarrow^* ID "+" ID "*" ID$

derivation

E
 $\Rightarrow E "+" E$
 $\Rightarrow ID "+" E$
 $\Rightarrow ID "+" E "*" E$
 $\Rightarrow ID "+" ID "*" E$
 $\Rightarrow ID "+" ID "*" ID$

derivation

E
 $\Rightarrow E "*" E$
 $\Rightarrow E "+" E "*" E$
 $\Rightarrow ID "+" E "*" E$
 $\Rightarrow ID "+" ID "*" E$
 $\Rightarrow ID "+" ID "*" ID$

Ambiguity: Deriving Abstract Syntax Terms

grammar

productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.N = "-" E$
 $E.P = "(" E ")"$
 $E.V = ID$

derivation

$E \Rightarrow^* ID "+" ID "*" ID$

derivation

E
 $\Rightarrow E "+" E$
 $\Rightarrow ID "+" E$
 $\Rightarrow ID "+" E "*" E$
 $\Rightarrow ID "+" ID "*" E$
 $\Rightarrow ID "+" ID "*" ID$

term derivation

E
 $\Rightarrow A(E, E)$
 $\Rightarrow A(V(ID), E)$
 $\Rightarrow A(V(ID), T(E, E))$
 $\Rightarrow A(V(ID), T(V(ID), E))$
 $\Rightarrow A(V(ID), T(V(ID), V(ID)))$

derivation

E
 $\Rightarrow E "*" E$
 $\Rightarrow E "+" E "*" E$
 $\Rightarrow ID "+" E "*" E$
 $\Rightarrow ID "+" ID "*" E$
 $\Rightarrow ID "+" ID "*" ID$

Ambiguity: Deriving Abstract Syntax Terms

grammar

productions

$E \cdot A = E \ " + " \ E$
 $E \cdot T = E \ " * " \ E$
 $E \cdot N = " - " \ E$
 $E \cdot P = "(" \ E \ ")"$
 $E \cdot V = ID$

derivation

$E \xrightarrow{*} ID \ " + " \ ID \ " * " \ ID$

derivation

E
 $\Rightarrow E \ " + " \ E$
 $\Rightarrow ID \ " + " \ E$
 $\Rightarrow ID \ " + " \ E \ " * " \ E$
 $\Rightarrow ID \ " + " \ ID \ " * " \ E$
 $\Rightarrow ID \ " + " \ ID \ " * " \ ID$

term derivation

E
 $\Rightarrow A(E, E)$
 $\Rightarrow A(V(ID), E)$
 $\Rightarrow A(V(ID), T(E, E))$
 $\Rightarrow A(V(ID), T(V(ID), E))$
 $\Rightarrow A(V(ID), T(V(ID), V(ID)))$

derivation

E
 $\Rightarrow E \ " * " \ E$
 $\Rightarrow E \ " + " \ E \ " * " \ E$
 $\Rightarrow ID \ " + " \ E \ " * " \ E$
 $\Rightarrow ID \ " + " \ ID \ " * " \ E$
 $\Rightarrow ID \ " + " \ ID \ " * " \ ID$

term derivation

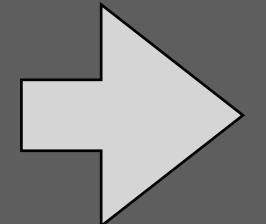
E
 $\Rightarrow T(E, E)$
 $\Rightarrow T(A(E, E), E)$
 $\Rightarrow T(A(V(ID), E), E)$
 $\Rightarrow T(A(V(ID), V(ID)), E)$
 $\Rightarrow T(A(V(ID), V(ID)), V(ID))$

Disambiguation

Traditional: Ambiguity = Parse Table Conflict

context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>>  
  
Exp.Fun     = <function(<{ID "," }*>) <Exp>>  
Exp.App     = <<Exp> <Exp>>  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd    = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>>
```



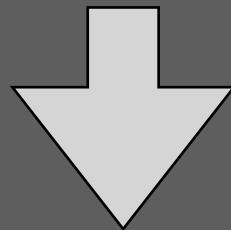
No can parse

Ambiguity = Multiple Possible Parses

context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>>  
  
Exp.Fun     = <function(<{ID "," }*>) <Exp>>  
Exp.App     = <<Exp> <Exp>>  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
Bnd.Bnd     = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>>
```

a + b + c



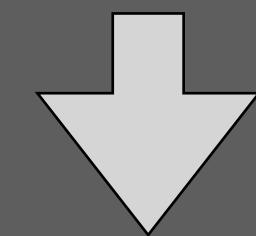
```
amb(  
  [ Add(Var("a"), Add(Var("b"), Var("c")))  
  , Add(Add(Var("a"), Var("b")), Var("c"))  
  ]  
)
```

Disambiguation = Select(Structure)

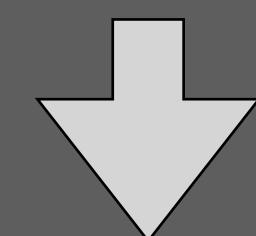
context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>>  
  
Exp.Fun     = <function(<{ID "," }*>) <Exp>>  
Exp.App     = <<Exp> <Exp>>  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
Bnd.Bnd     = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>>
```

a + b + c



```
amb(  
  [ Add(Var("a"), Add(Var("b"), Var("c")))  
  , Add(Add(Var("a"), Var("b")), Var("c"))  
  ]  
)
```



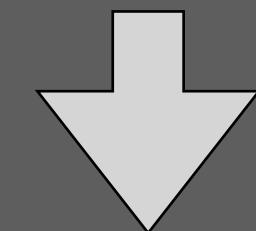
Add(Add(Var("a"), Var("b")), Var("c"))

Brackets = Explicit Disambiguation

context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>>  
  
Exp.Fun     = <function(<{ID ","}>*)> <Exp>>  
Exp.App     = <<Exp> <Exp>>  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
Bnd.Bnd     = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>>
```

a + (b + c)

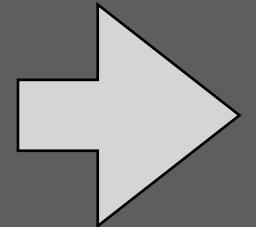


Add(Var("a"), Add(Var("b"), Var("c")))

Disambiguation by Manual Transformation = Bad

context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>>  
  
Exp.Fun     = <function(<{ID "," }*>) <Exp>>  
Exp.App     = <<Exp> <Exp>>  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd    = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>>
```



Big ugly grammar

Declarative Disambiguation = Separate Concern

context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>> {left}  
  
Exp.Fun     = <function(<{ID ","}*>) <Exp>>  
Exp.App     = <<Exp> <Exp>> {left}  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd    = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
               {longest-match}  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>> {left}
```

context-free priorities

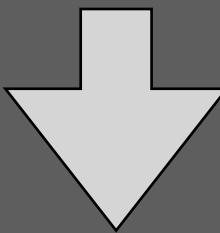
```
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

Associativity = Solve Intra Operator Ambiguity

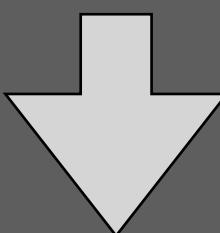
context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int      = INT  
Exp.Var      = ID  
Exp.Add      = <<Exp> + <Exp>> {left}  
  
Exp.Fun      = <function(<{ID ","}*>) <Exp>>  
Exp.App      = <<Exp> <Exp>> {left}  
  
Exp.Let      = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd      = <<ID> = <Exp>>  
  
Exp.If        = <if(<Exp>) <Exp>>  
Exp.IfElse    = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match    = <match <Exp> with <{Case " | "}>+>  
                  {longest-match}  
Case.Case    = [[Pat] → [Exp]]  
  
Pat.PVar     = ID  
Pat.PApp     = <<Pat> <Pat>> {left}  
  
context-free priorities  
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

a + b + c



```
amb(  
  [ Add(Var("a"), Add(Var("b"), Var("c")))  
  , Add(Add(Var("a"), Var("b")), Var("c"))  
  ]  
)
```



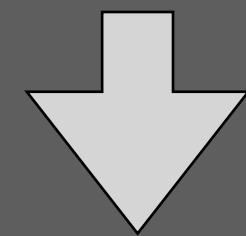
Add(Add(Var("a"), Var("b")), Var("c"))

Priority = Solve Inter Operator Ambiguity

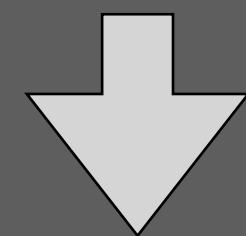
context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>> {left}  
  
Exp.Fun     = <function(<{ID ","}*>) <Exp>>  
Exp.App     = <<Exp> <Exp>> {left}  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd    = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
               {longest-match}  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>> {left}  
  
context-free priorities  
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

f a + b



```
amb(  
  [ Add(App(Var("f")), Var("a")), Var("b"))  
  , App(Var("f")), Add(Var("a"), Var("b")))  
)
```



Add(App(Var("f")), Var("a")), Var("b"))

Dangling Else = Operators with Overlapping Prefix

context-free syntax

```
Exp      = <(<Exp>)> {bracket}  
  
Exp.Int  = INT  
Exp.Var  = ID  
Exp.Add  = <<Exp> + <Exp>> {left}  
  
Exp.Fun  = <function(<{ID ","}>*)> <Exp>>  
Exp.App  = <<Exp> <Exp>> {left}  
  
Exp.Let  = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd  = <<ID> = <Exp>>  
  
Exp.If   = <if(<Exp>) <Exp>>  
Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match = <match <Exp> with <{Case " | "}>+>  
           {longest-match}  
Case.Case = [[Pat] → [Exp]]  
  
Pat.PVar  = ID  
Pat.PApp  = <<Pat> <Pat>> {left}  
  
context-free priorities  
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

if(1) if(2) 3 else 4

amb([IfElse(
 Int("1")
 , If(Int("2"), Int("3"))
 , Int("4")
)
 , If(
 Int("1")
 , IfElse(Int("2"), Int("3"), Int("4"))
)
]

If(
 Int("1")
 , IfElse(Int("2"), Int("3"), Int("4"))
)

Safe Disambiguation = Do Not Reject Unambiguous Sentences

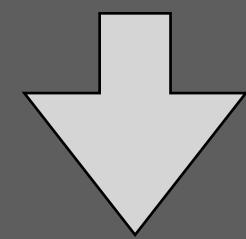
context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int      = INT  
Exp.Var      = ID  
Exp.Add      = <<Exp> + <Exp>> {left}  
  
Exp.Fun      = <function(<{ID ","}*>) <Exp>>  
Exp.App      = <<Exp> <Exp>> {left}  
  
Exp.Let      = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd      = <<ID> = <Exp>>  
  
Exp.If        = <if(<Exp>) <Exp>>  
Exp.IfElse    = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match    = <match <Exp> with <{Case " | "}>+>  
                  {longest-match}  
Case.Case    = [[Pat] → [Exp]]  
  
Pat.PVar     = ID  
Pat.PApp     = <<Pat> <Pat>> {left}
```

context-free priorities

```
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

4 + if(y) x



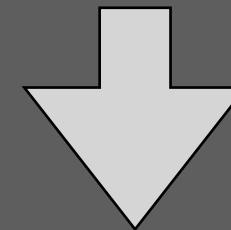
Add(Int("4"), If(Var("y"), Var("x")))

Deep Priority Conflict

context-free syntax

```
Exp      = <(<Exp>)> {bracket}  
  
Exp.Int   = INT  
Exp.Var   = ID  
Exp.Add   = <<Exp> + <Exp>> {left}  
  
Exp.Fun   = <function(<{ID ","}*>) <Exp>>  
Exp.App   = <<Exp> <Exp>> {left}  
  
Exp.Let   = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd   = <<ID> = <Exp>>  
  
Exp.If    = <if(<Exp>) <Exp>>  
Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match = <match <Exp> with <{Case " | "}>+>  
           {longest-match}  
Case.Case = [[Pat] → [Exp]]  
  
Pat.PVar  = ID  
Pat.PApp  = <<Pat> <Pat>> {left}  
  
context-free priorities  
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

4 + if(y) x + 3



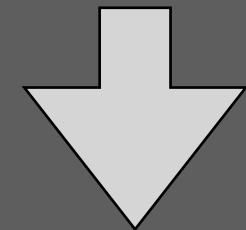
```
amb(  
  [ Add(  
    Int("4")  
  , amb(  
    [ Add(If(Var("y"), Var("x")), Int("3"))  
    , If(Var("y"), Add(Var("x"), Int("3")))  
    ]  
  )  
  , Add(  
    Add(Int("4"), If(Var("y"), Var("x")))  
    , Int("3")  
  )  
]  
)
```

Deep Priority Conflict (Solved)

context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>> {left}  
  
Exp.Fun     = <function(<{ID ","}*>) <Exp>>  
Exp.App     = <<Exp> <Exp>> {left}  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd    = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
               {longest-match}  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>> {left}  
  
context-free priorities  
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

4 + if(y) x + 3



```
Add(  
  Int("4")  
, If(Var("y"), Add(Var("x"), Int("3"))))  
)
```

Longest Match = Solve Repetition Ambiguity

context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>> {left}  
  
Exp.Fun     = <function(<{ID ","}>*)> <Exp>>  
Exp.App     = <<Exp> <Exp>> {left}  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd    = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
               {longest-match}  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>> {left}  
  
context-free priorities  
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

```
match x with  
  a → match 5 with  
    b → 3  
  | c → 4
```

```
Match(  
  Var("x"))  
, amb(  
  [ [ Case(  
        PVar("a"))  
    , Match(  
        Int("5"))  
    , [ Case(PVar("b"), Int("3"))]  
    ]  
  )  
  , Case(PVar("c"), Int("4"))  
  , [ Case(  
        PVar("a"))  
    , Match(  
        Int("5"))  
    , [ Case(PVar("b"), Int("3"))  
        , Case(PVar("c"), Int("4"))  
    ]  
  ]  
)
```

Longest Match = Solve Repetition Ambiguity

context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int     = INT  
Exp.Var     = ID  
Exp.Add     = <<Exp> + <Exp>> {left}  
  
Exp.Fun     = <function(<{ID ","}>*)> <Exp>>  
Exp.App     = <<Exp> <Exp>> {left}  
  
Exp.Let     = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd    = <<ID> = <Exp>>  
  
Exp.If      = <if(<Exp>) <Exp>>  
Exp.IfElse  = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match   = <match <Exp> with <{Case " | "}>+>  
              {longest-match}  
Case.Case   = [[Pat] → [Exp]]  
  
Pat.PVar    = ID  
Pat.PApp    = <<Pat> <Pat>> {left}  
  
context-free priorities  
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

```
match x with  
  a → match 5 with  
    b → 3  
  | c → 4
```

```
Match(  
  Var("x")  
, [ Case(  
    PVar("a")  
, Match(  
      Int("5")  
, [ Case(PVar("b"), Int("3"))  
      , Case(PVar("c"), Int("4"))  
    ]  
)  
)  
)  
)
```

Parenthesize

Parenthesize = Disambiguate⁻¹ (Insert Necessary Parentheses)

context-free syntax

Exp = <(<Exp>)> {bracket}

Exp.Int = INT

Exp.Var = ID

Exp.Add = <<Exp> + <Exp>> {left}

Exp.Fun = <function(<{ID ","}*>) <Exp>>

Exp.App = <<Exp> <Exp>> {left}

Exp.Let = <let <Bnd*> in <Exp>>

Bnd.Bnd = <<ID> = <Exp>>

Exp.If = <if(<Exp>) <Exp>>

Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

Exp.Match = <match <Exp> with <{Case " | "}>+>
{longest-match}

Case.Case = [[Pat] → [Exp]]

Pat.PVar = ID

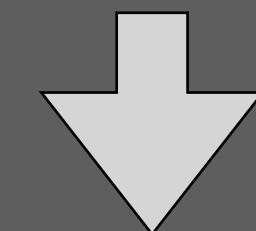
Pat.PApp = <<Pat> <Pat>> {left}

context-free priorities

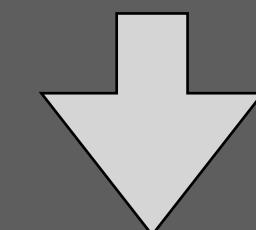
Exp.App > Exp.Add > Exp.IfElse > Exp.If

> Exp.Match > Exp.Let > Exp.Fun

(a + b) + c



Add(Add(Var("a"), Var("b")), Var("c"))



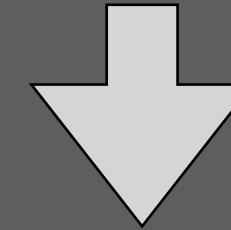
a + b + c

Parenthesize = Disambiguate⁻¹ (Insert Necessary Parentheses)

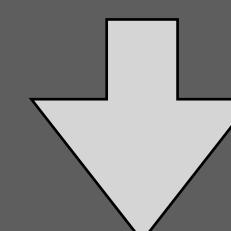
context-free syntax

```
Exp          = <(<Exp>)> {bracket}  
  
Exp.Int      = INT  
Exp.Var      = ID  
Exp.Add      = <<Exp> + <Exp>> {left}  
  
Exp.Fun      = <function(<{ID ","}>*)> <Exp>>  
Exp.App      = <<Exp> <Exp>> {left}  
  
Exp.Let      = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd      = <<ID> = <Exp>>  
  
Exp.If        = <if(<Exp>) <Exp>>  
Exp.IfElse    = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match    = <match <Exp> with <{Case " | "}>+>  
               {longest-match}  
Case.Case    = [[Pat] → [Exp]]  
  
Pat.PVar     = ID  
Pat.PApp     = <<Pat> <Pat>> {left}  
  
context-free priorities  
Exp.App > Exp.Add > Exp.IfElse > Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

a + (let x = b in (c + d))



```
Add(  
  Var("a")  
, Let(  
  [Bnd("x", Var("b"))]  
, Add(Var("c"), Var("d"))  
)  
)
```



a + let
 x = b
in
 c + d

Parenthesize = Disambiguate⁻¹ (Insert Necessary Parentheses)

context-free syntax

Exp = <(<Exp>)> {bracket}

Exp.Int = INT

Exp.Var = ID

Exp.Add = <<Exp> + <Exp>> {left}

Exp.Fun = <function(<{ID ","}>*)> <Exp>>

Exp.App = <<Exp> <Exp>> {left}

Exp.Let = <let <Bnd*> in <Exp>>

Bnd.Bnd = <<ID> = <Exp>>

Exp.If = <if(<Exp>) <Exp>>

Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>

Exp.Match = <match <Exp> with <{Case " | "}>+>
{longest-match}

Case.Case = [[Pat] → [Exp]]

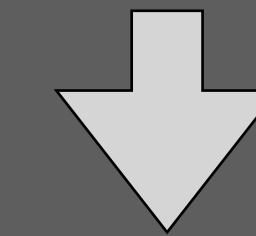
Pat.PVar = ID

Pat.PApp = <<Pat> <Pat>> {left}

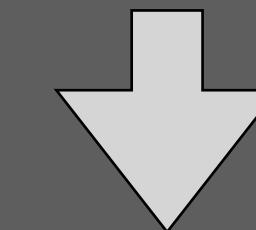
context-free priorities

Exp.App > Exp.Add > Exp.IfElse > Exp.If
> Exp.Match > Exp.Let > Exp.Fun

(a + (let x = b in c)) + d



Add(
 Add(
 Var("a")
 , Let([Bnd("x", Var("b"))], Var("c"))
)
 , Var("d")
)



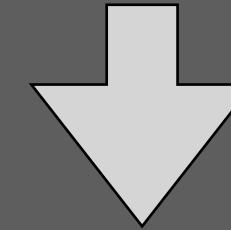
a + (let
 x = b
 in
 c) + d

Parenthesize = Disambiguate⁻¹ (Insert Necessary Parentheses)

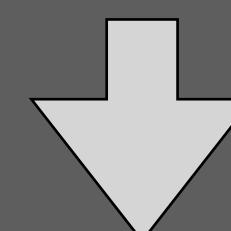
context-free syntax

```
Exp      = <(<Exp>)> {bracket}  
  
Exp.Int  = INT  
Exp.Var  = ID  
Exp.Add   = <<Exp> + <Exp>> {left}  
Exp.Sub   = <<Exp> - <Exp>> {left}  
  
Exp.Fun   = <function(<{ID ","}>*)> <Exp>>  
Exp.App   = <<Exp> <Exp>> {left}  
  
Exp.Let   = <let <Bnd*> in <Exp>>  
  
Bnd.Bnd  = <<ID> = <Exp>>  
  
Exp.If    = <if(<Exp>) <Exp>>  
Exp.IfElse = <if(<Exp>) <Exp> else <Exp>>  
  
Exp.Match = <match <Exp> with <{Case "|"}+>>  
            {longest-match}  
Case.Case = [[Pat] → [Exp]]  
  
Pat.PVar  = ID  
Pat.PApp  = <<Pat> <Pat>> {left}  
  
context-free priorities  
{left: Exp.Sub Exp.App} > Exp.Add > Exp.IfElse >  
Exp.If  
> Exp.Match > Exp.Let > Exp.Fun
```

(a + (let x = b in c)) + d



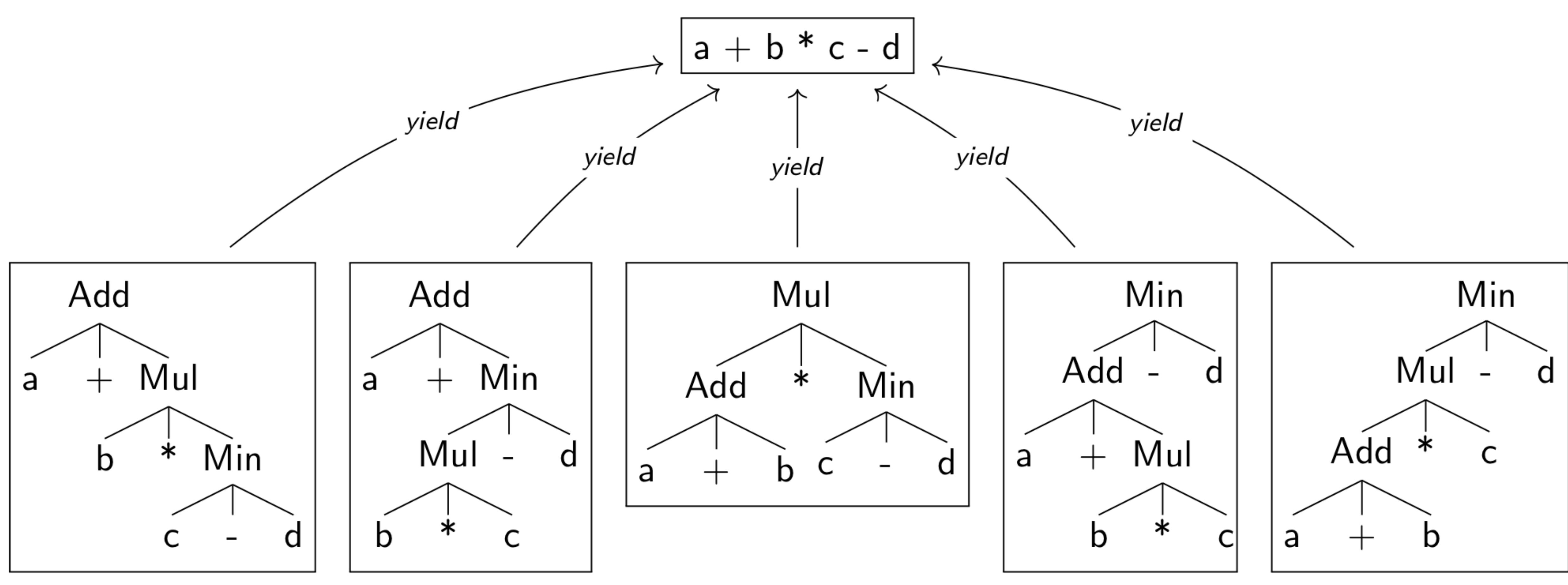
```
Add(  
  Add(  
    Var("a")  
  , Let([Bnd("x", Var("b"))], Var("c"))  
  )  
, Var("d"))  
)
```



a + (let
 x = b
 in
 c) + d

Semantics of Associativity and Priority

Ambiguous Sentence has Multiple Parse Trees



Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --- alpha["\u03b1"]; C1 --- C2["C\u2082"]; C1 --- gamma["\u03b3"]; C2 --- beta["\u03b2"];
```

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --- alpha["\u03b1"]; C1 --- C2["C\u2082"]; C1 --- gamma["\u03b3"]; C2 --- beta["\u03b2"];
```

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Instances

Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --- alpha; C1 --- C2; C1 --- gamma; C2 --- beta;
```

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Instances

$$\frac{E.Mul > E.Add}{Mul}$$

```
graph TD; Mul --- Add; Mul --- Star["*"]; Mul --- E1[E]; Add --- E2[E]; Add --- Plus["+"]; Plus --- E3[E]; Plus --- E4[E];
```

Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --- alpha; C1 --- C2; C1 --- gamma; C2 --- beta;
```

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Instances

$$\frac{E.Mul > E.Add}{Mul}$$

```
graph TD; Mul --- Add; Mul --- E1[E]; Add --- E2[E]; Add --- E3[E];
```

$$\frac{E.Mul > E.Add}{Mul}$$

```
graph TD; Mul --- E1[E]; Mul --- Add; Add --- E2[E]; Add --- E3[E];
```

Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --- alpha; C1 --- C2; C2 --- beta; C2 --- gamma;
```

$$\frac{A.C_1 \text{ left } A.C_2}{C_1}$$

```
graph TD; C1 --- alpha; C1 --- C2; C2 --- beta;
```

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Instances

$$\frac{E.Mul > E.Add}{Mul}$$

```
graph TD; Mul --- Add; Mul --- star; Add --- E1; Add --- E2;
```

$$\frac{E.Mul > E.Add}{Mul}$$

```
graph TD; Mul --- E1; Mul --- star; star --- Add; star --- E3;
```

Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

A diagram showing a tree node labeled C_1 with three children: α , C_2 , and γ . The node C_2 has one child, β .

$$\frac{A.C_1 \text{ left } A.C_2}{C_1}$$

A diagram showing a tree node labeled C_1 with two children: α and C_2 . The node C_2 has one child, β .

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Instances

$$\frac{E.Mul > E.Add}{Mul}$$

A diagram showing a tree node labeled Mul with three children: Add , $*$, and E . The node Add has two children: E and E .

$$\frac{E.Mul > E.Add}{Mul}$$

A diagram showing a tree node labeled Mul with three children: E , $*$, and Add . The node Add has two children: E and E .

$$\frac{E.Add \text{ left } E.Add}{Add}$$

A diagram showing a tree node labeled Add with two children: E and Add . The node Add has two children: E and E .

Associativity and Priority as Subtree Exclusion Rules [SDF2 (1997)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --- alpha; C1 --- C2; C2 --- beta; C2 --- gamma;
```

$$\frac{A.C_1 \text{ left } A.C_2}{C_1}$$

```
graph TD; C1 --- alpha; C1 --- C2; C2 --- beta;
```

$$\frac{A.C_1 \text{ right } A.C_2}{C_1}$$

```
graph TD; C1 --- C2; C1 --- gamma; C2 --- beta;
```

Disambiguation rules generate subtree exclusion patterns (aka conflict patterns)

Instances

$$\frac{E.Mul > E.Add}{Mul}$$

```
graph TD; Mul --- Add; Mul --- star; Add --- E1; Add --- E2; E1 --- E3; E1 --- E4;
```

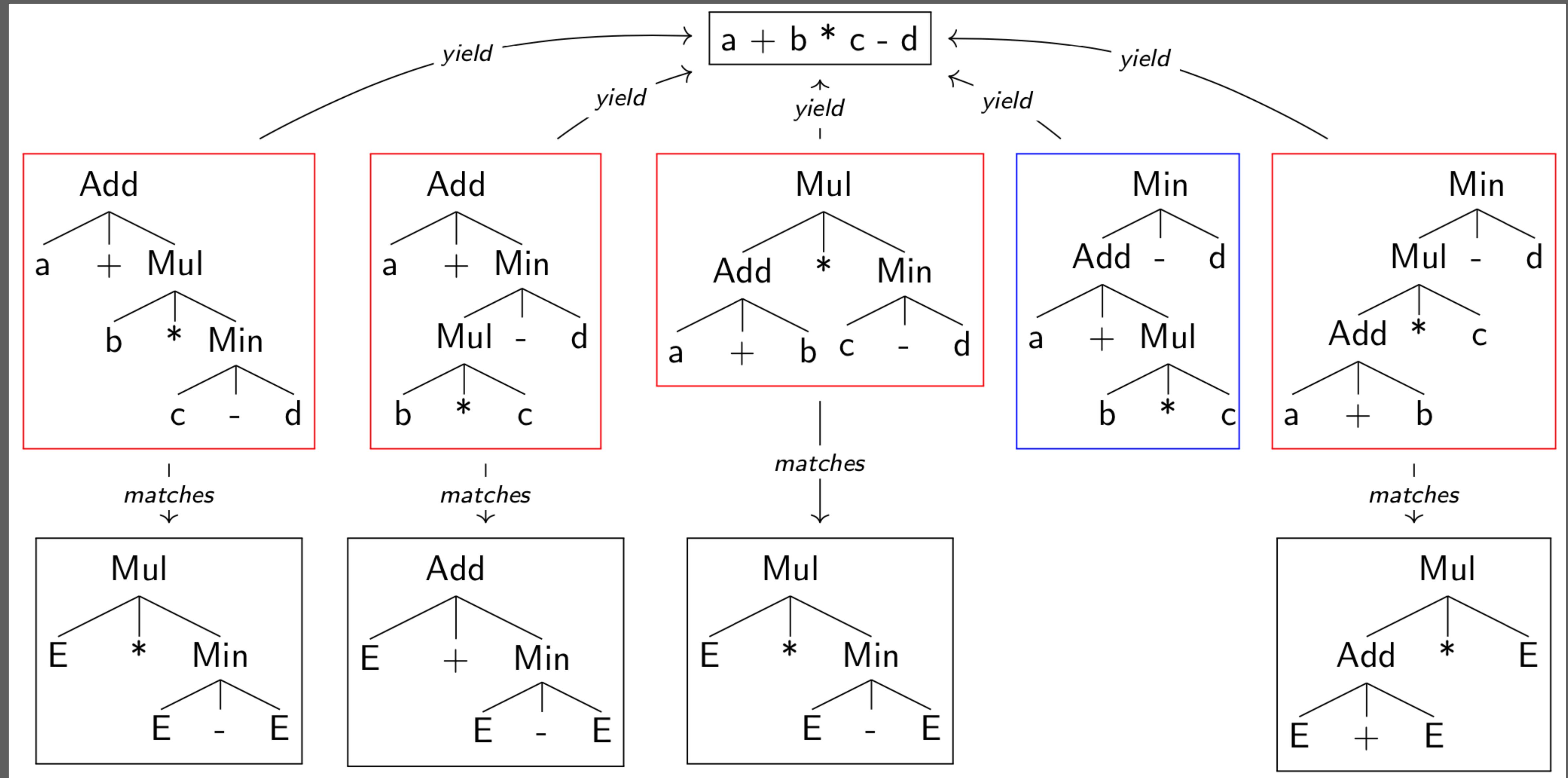
$$\frac{E.Mul > E.Add}{Mul}$$

```
graph TD; Mul --- E1; Mul --- star; Mul --- Add; E1 --- E3; E1 --- E4; star --- E5; star --- E6;
```

$$\frac{E.Add \text{ left } E.Add}{Add}$$

```
graph TD; Add --- E1; Add --- Add2; E1 --- E3; E1 --- E4; Add2 --- E5; Add2 --- E6;
```

Disambiguation by Subtree Exclusion



Safe for High Priority Prefix Operators

Conflict
Patterns

$$\frac{E.\text{Min} > E.\text{Add}}{\text{Min}}$$

```
graph TD; Min --> Add; Add --> E1[E]; Add --> E2[E]
```

Trees

Safe for High Priority Prefix Operators

Conflict
Patterns

$$\frac{E.\text{Min} > E.\text{Add}}{\text{Min}}$$

Add

```
graph TD; Add[Add] --- E1[E]; Add --- E2[E]
```

$$\frac{E.\text{Min} > E.\text{Mul}}{\text{Min}}$$

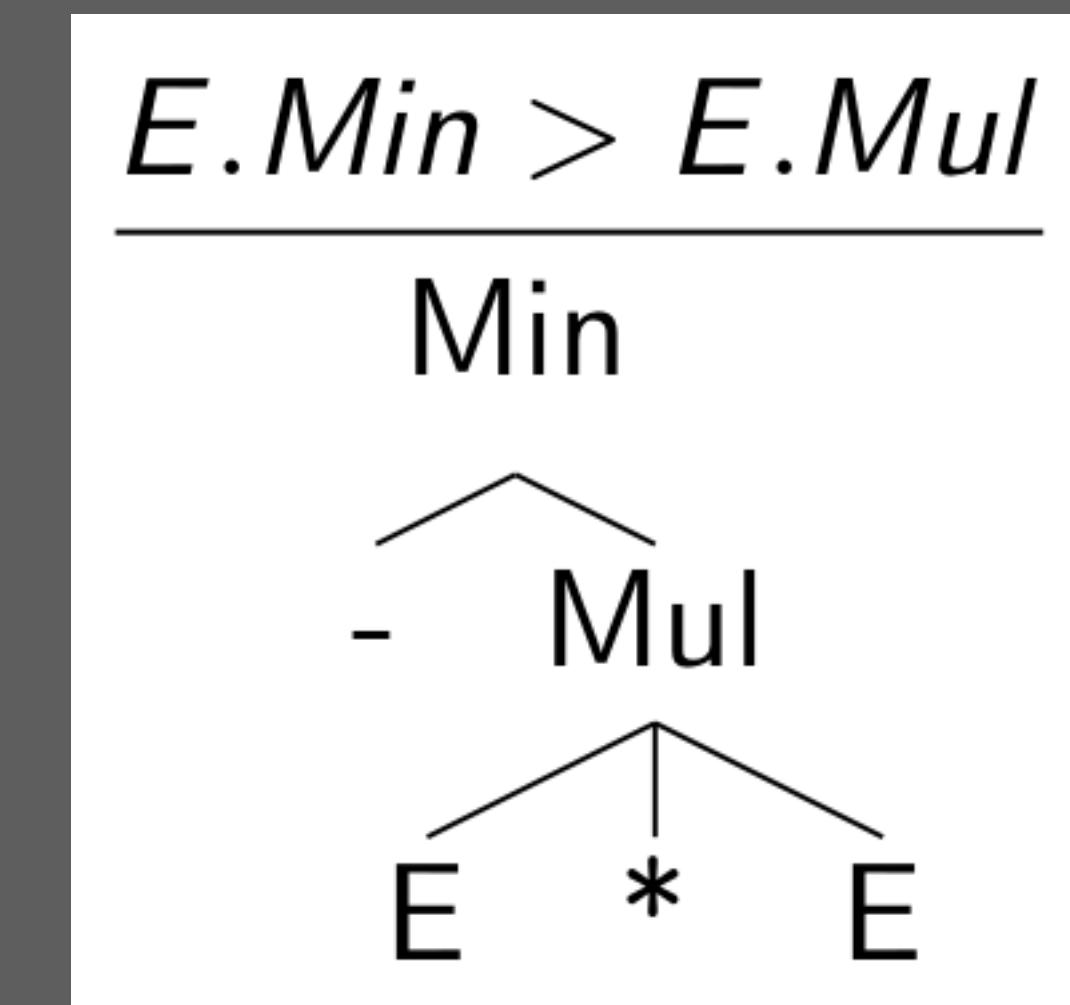
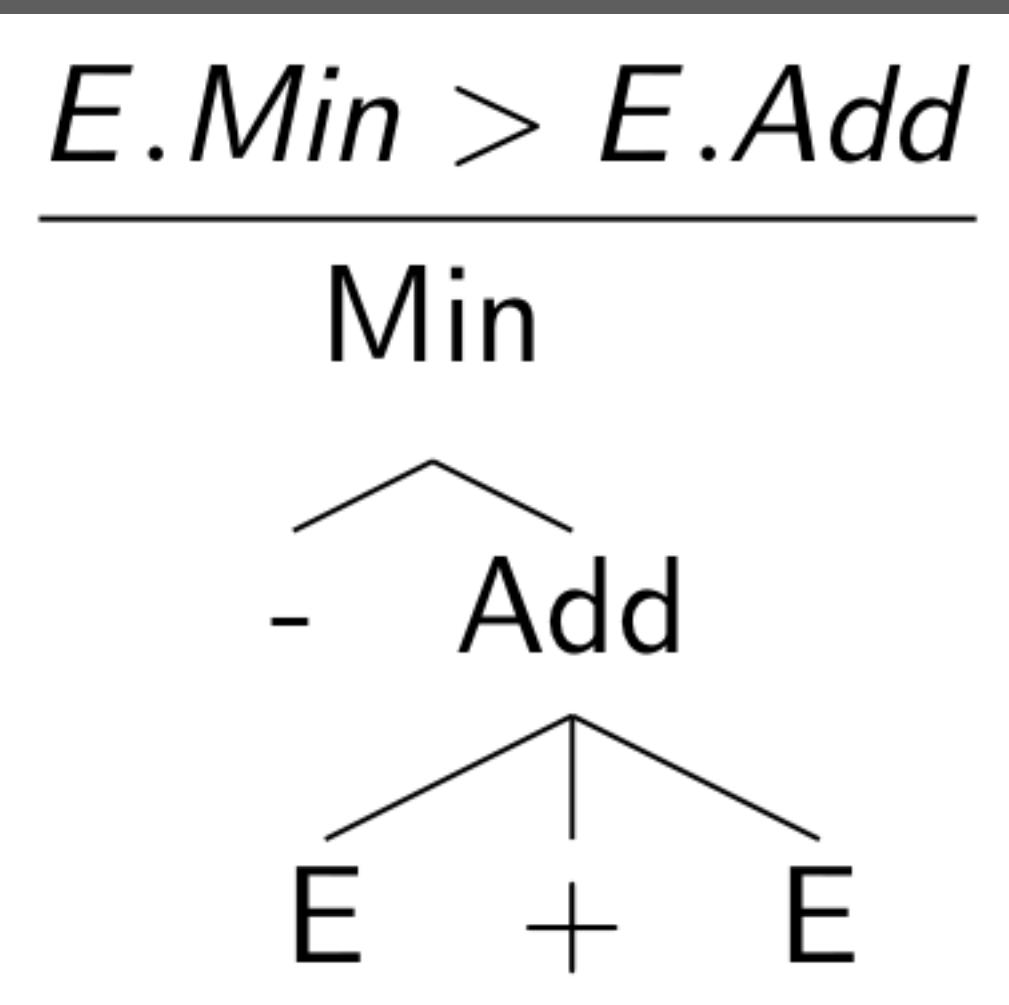
Mul

```
graph TD; Mul[Mul] --- E3[E]; Mul --- E4[E]
```

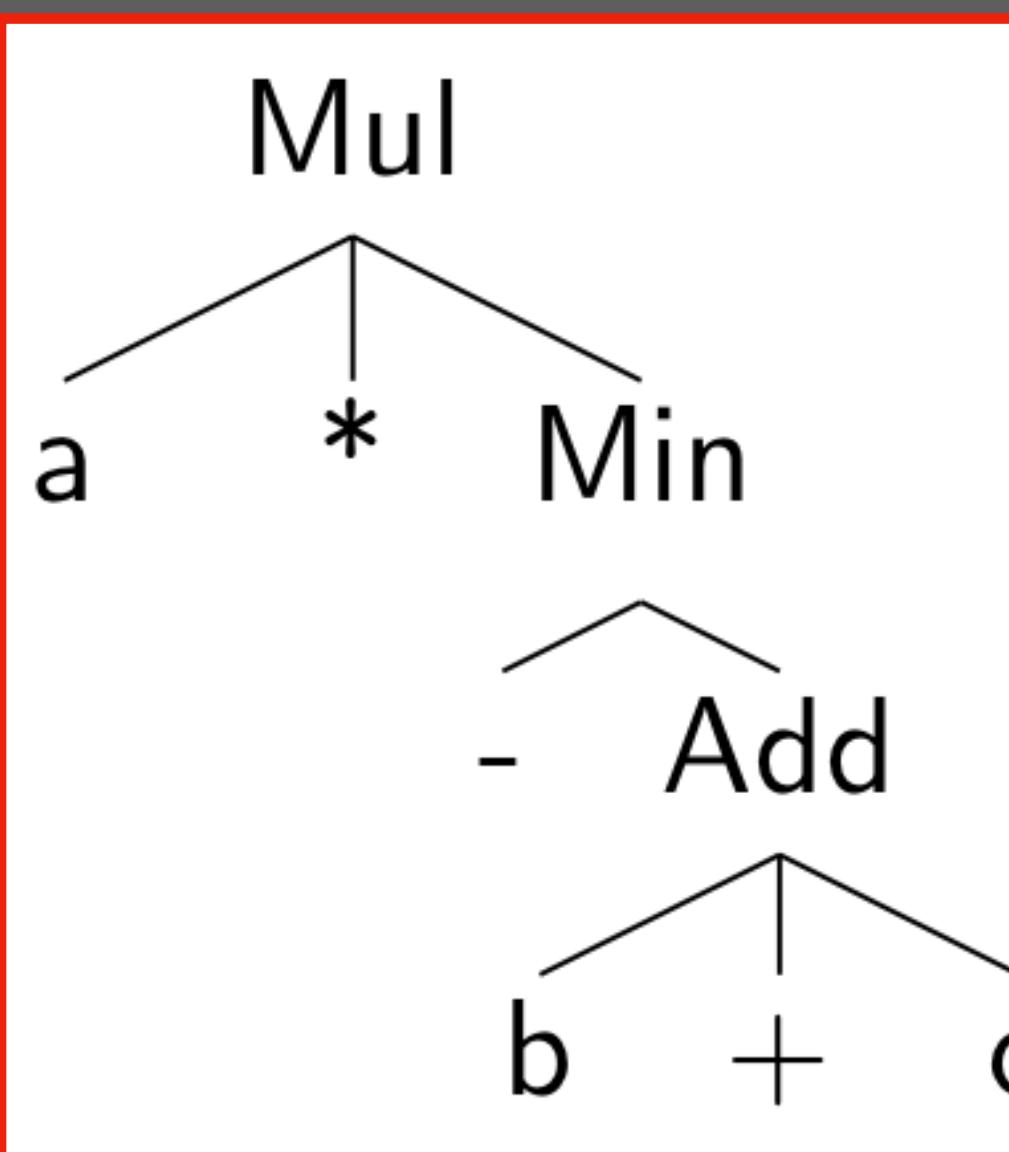
Trees

Safe for High Priority Prefix Operators

Conflict Patterns

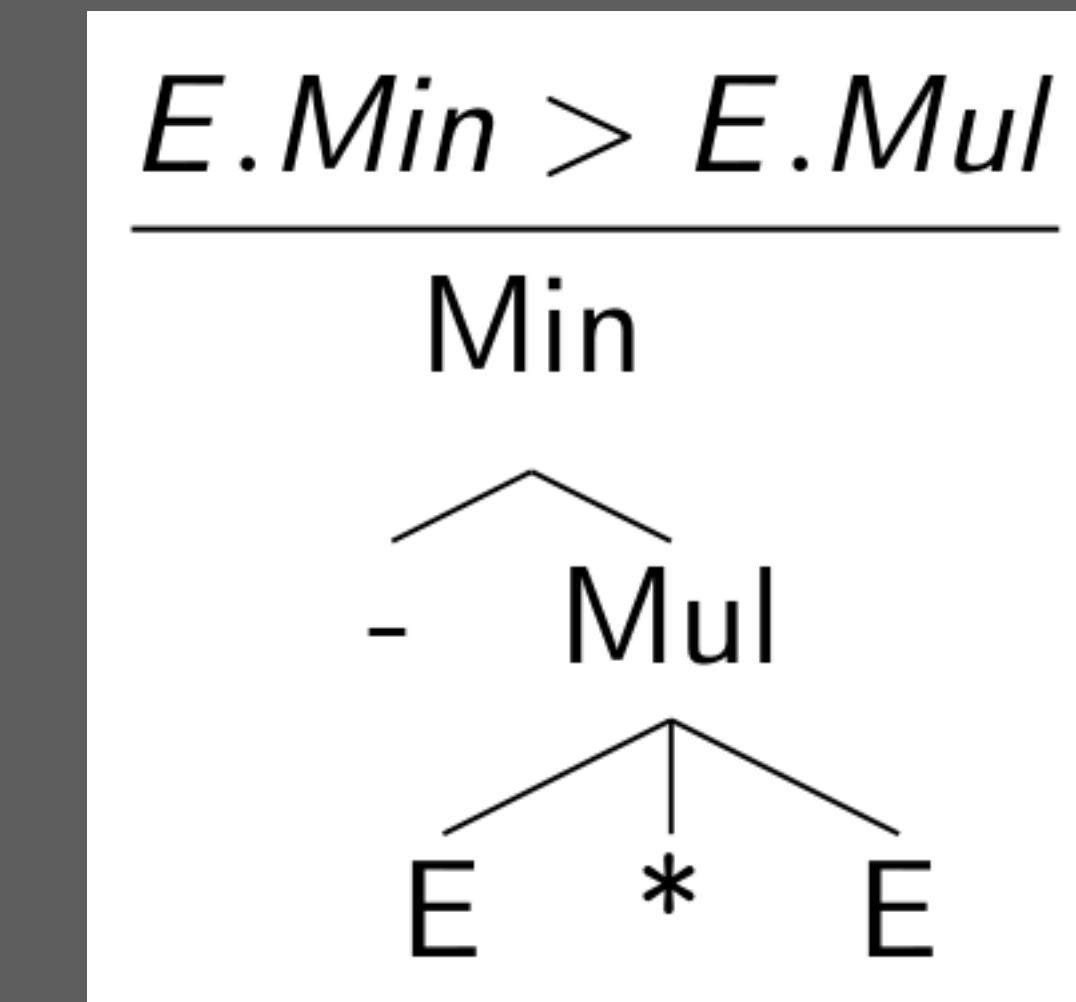
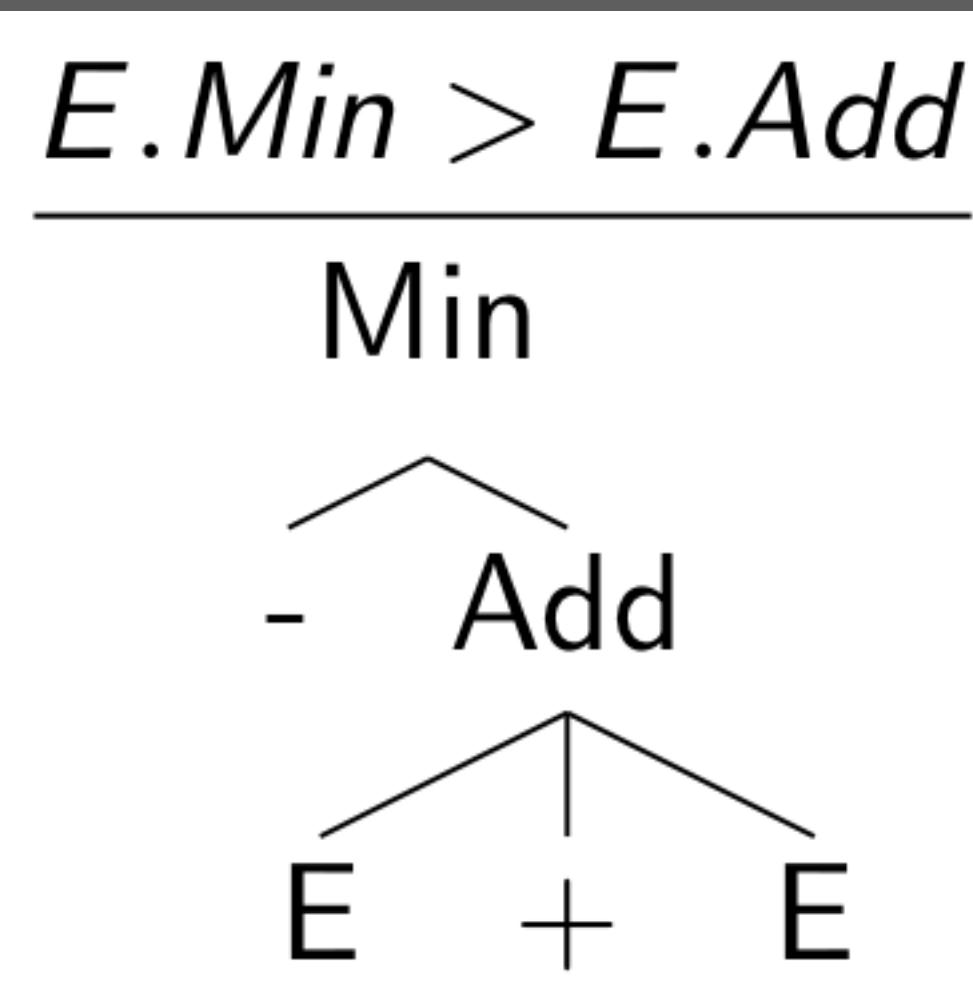


Trees

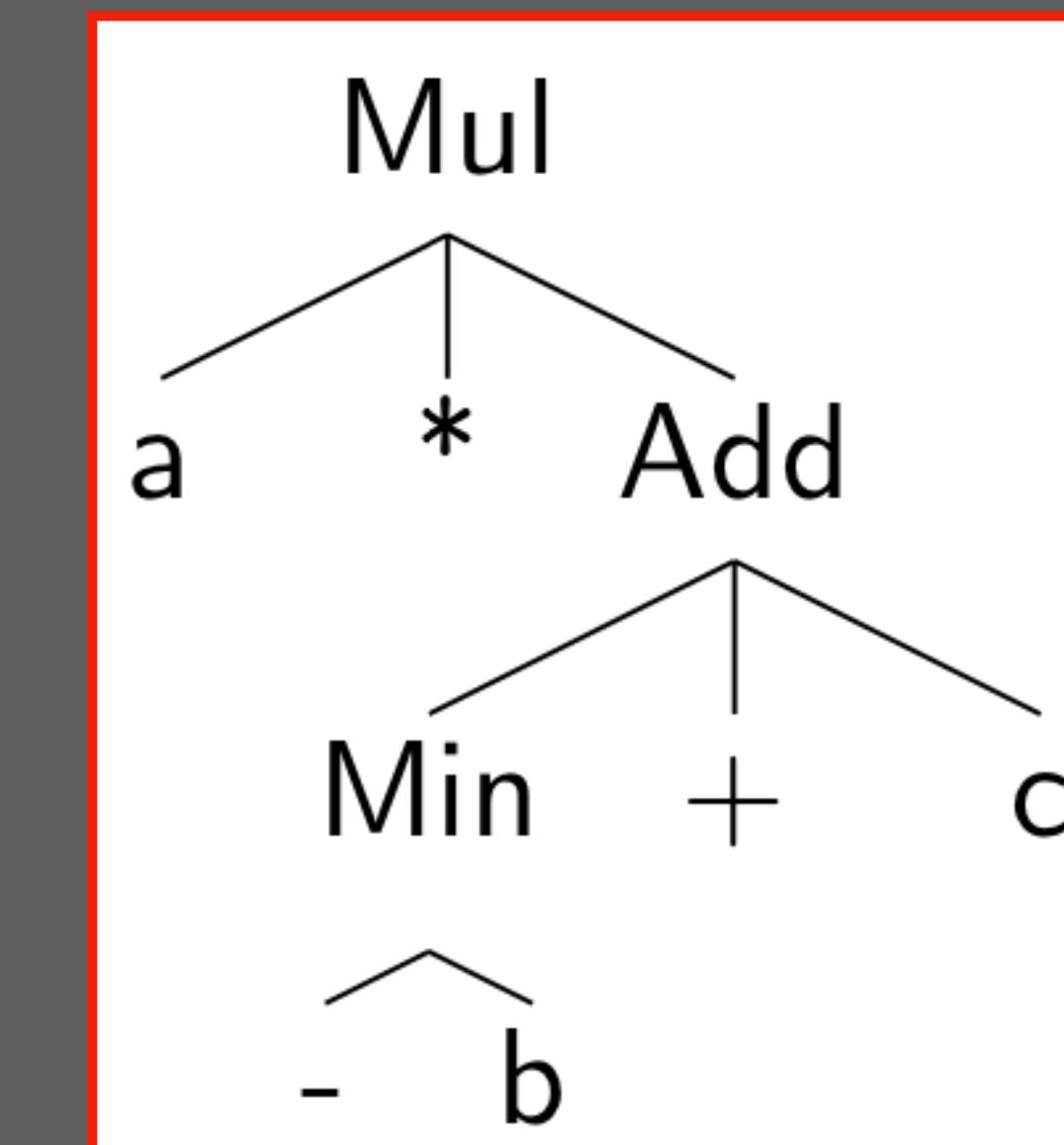
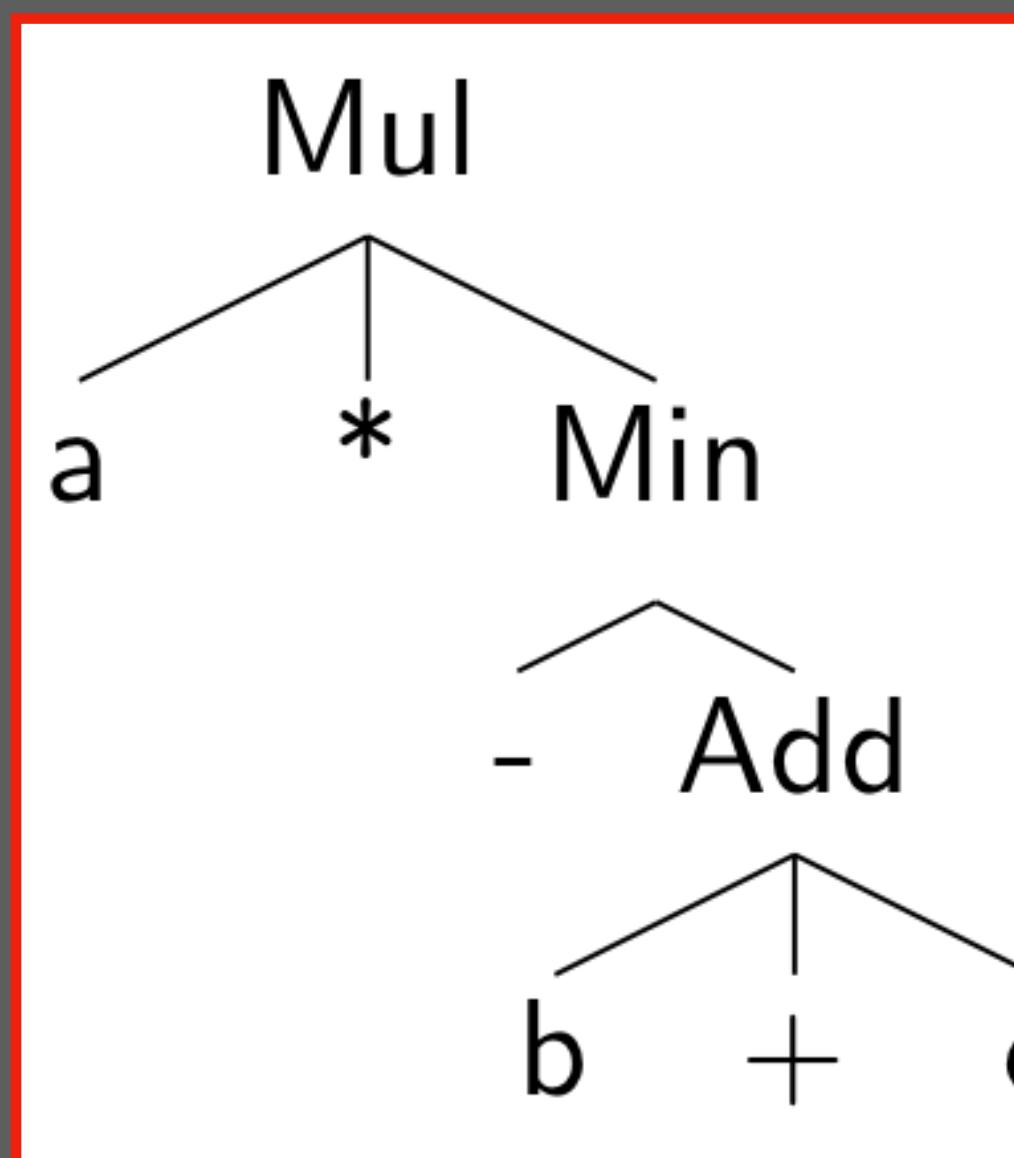


Safe for High Priority Prefix Operators

Conflict Patterns

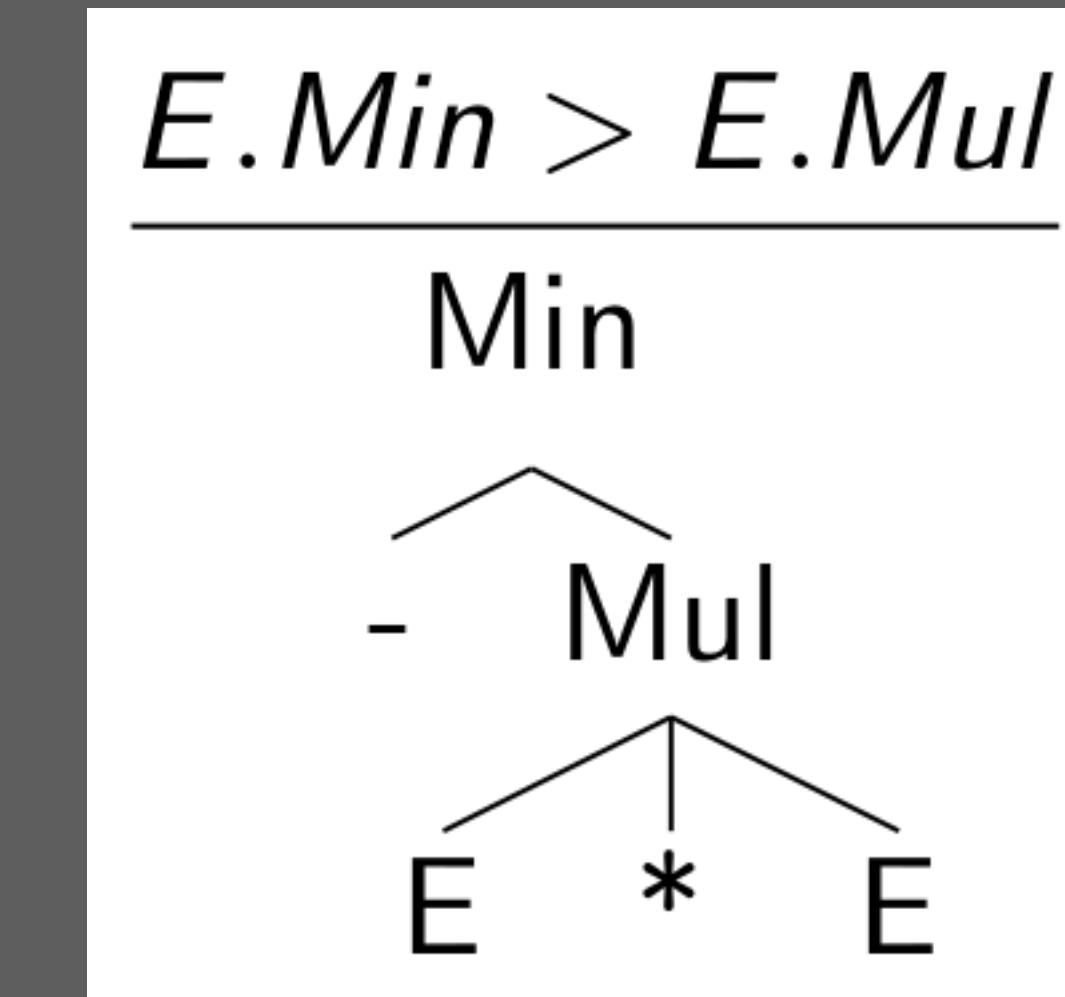
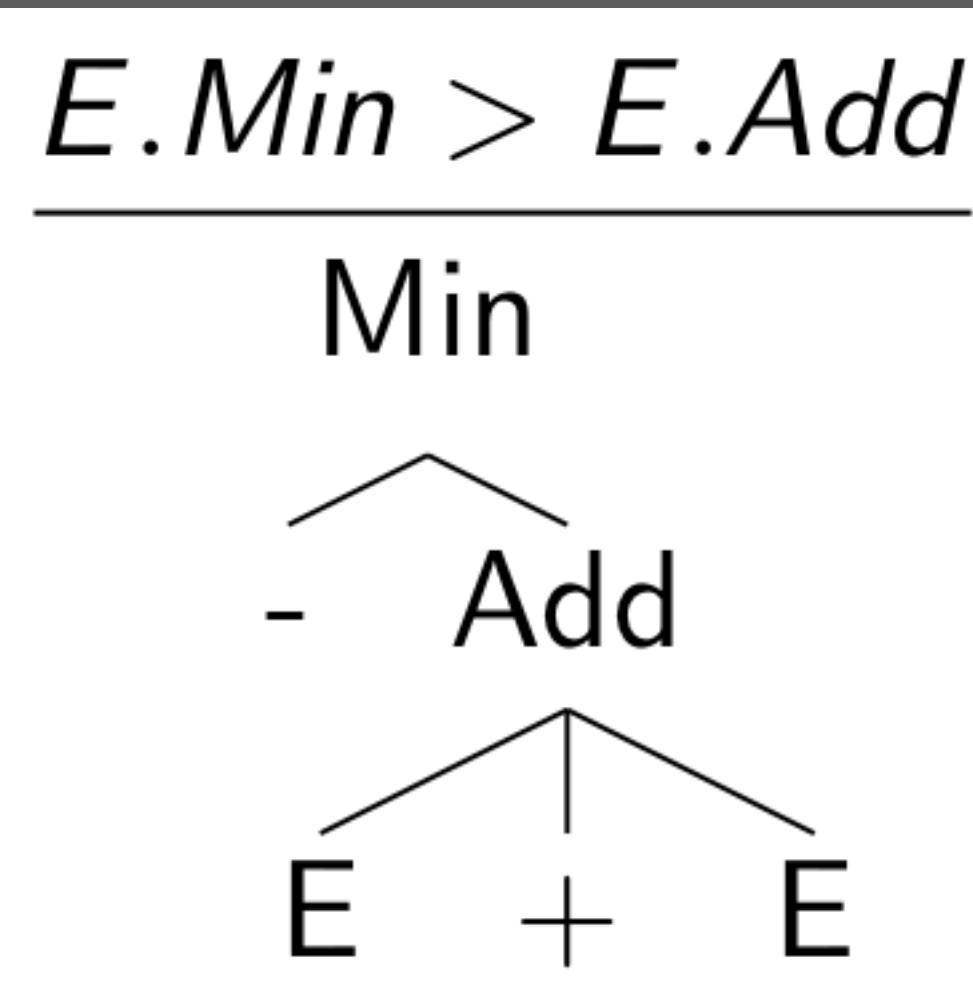


Trees

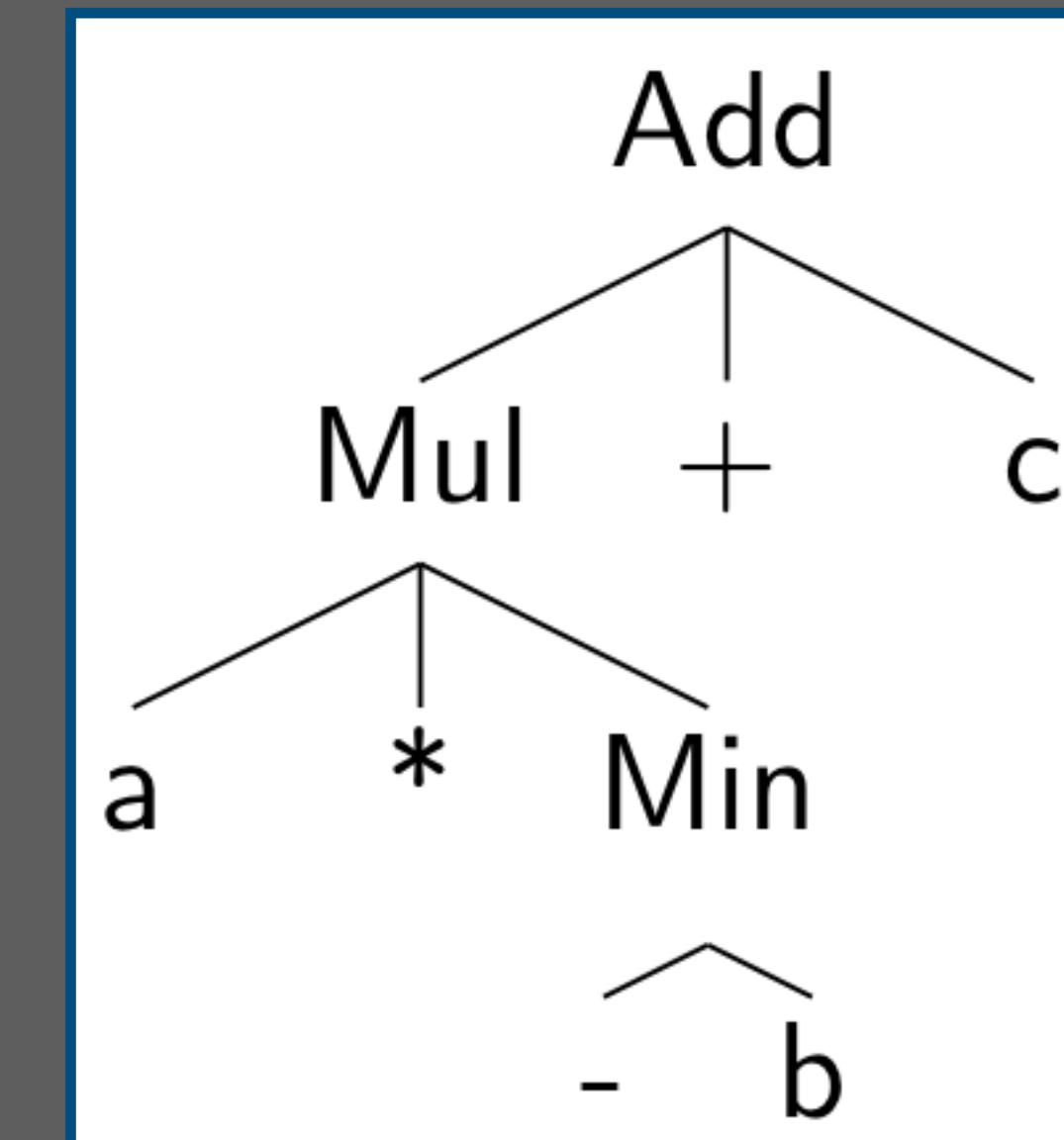
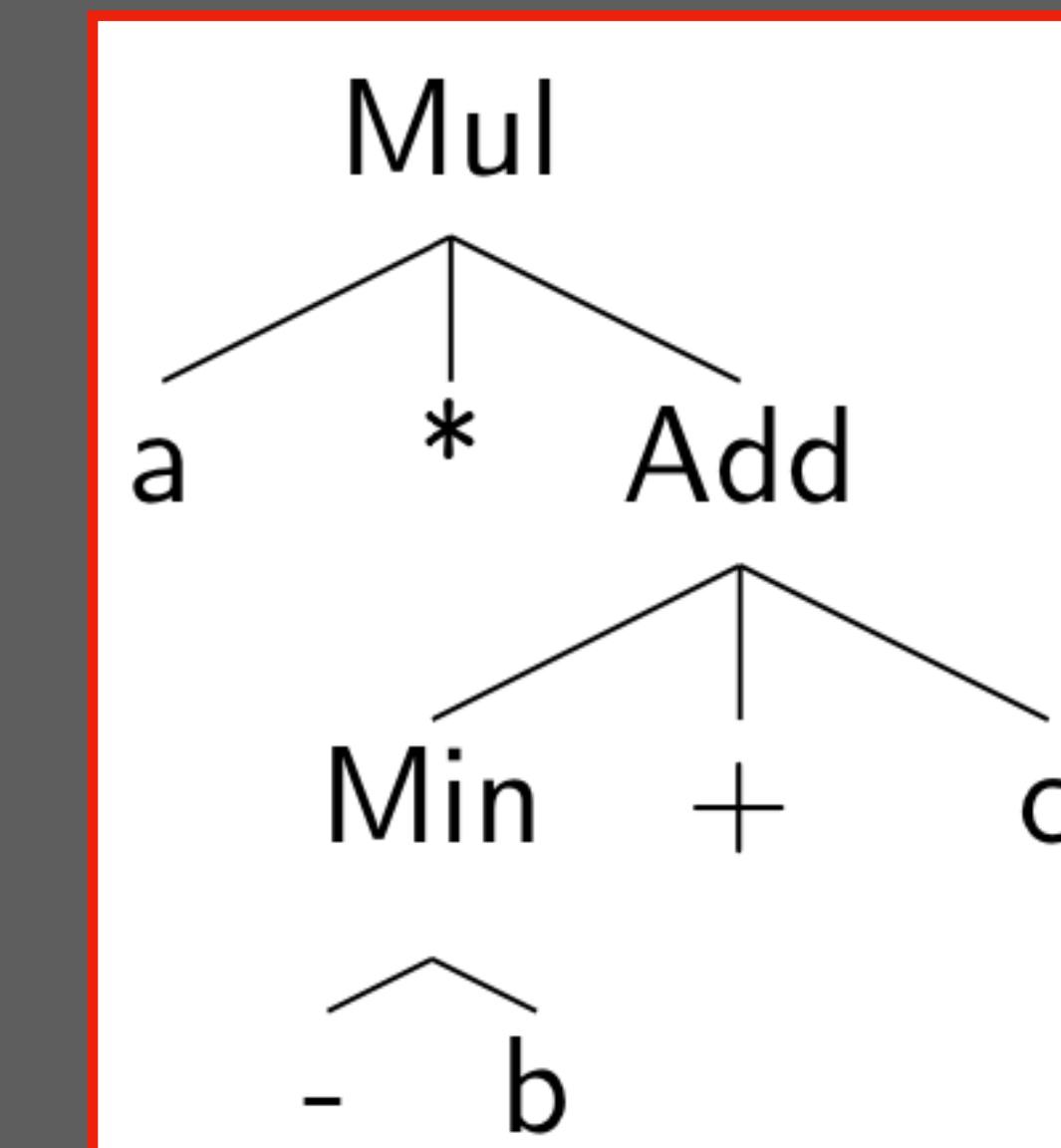
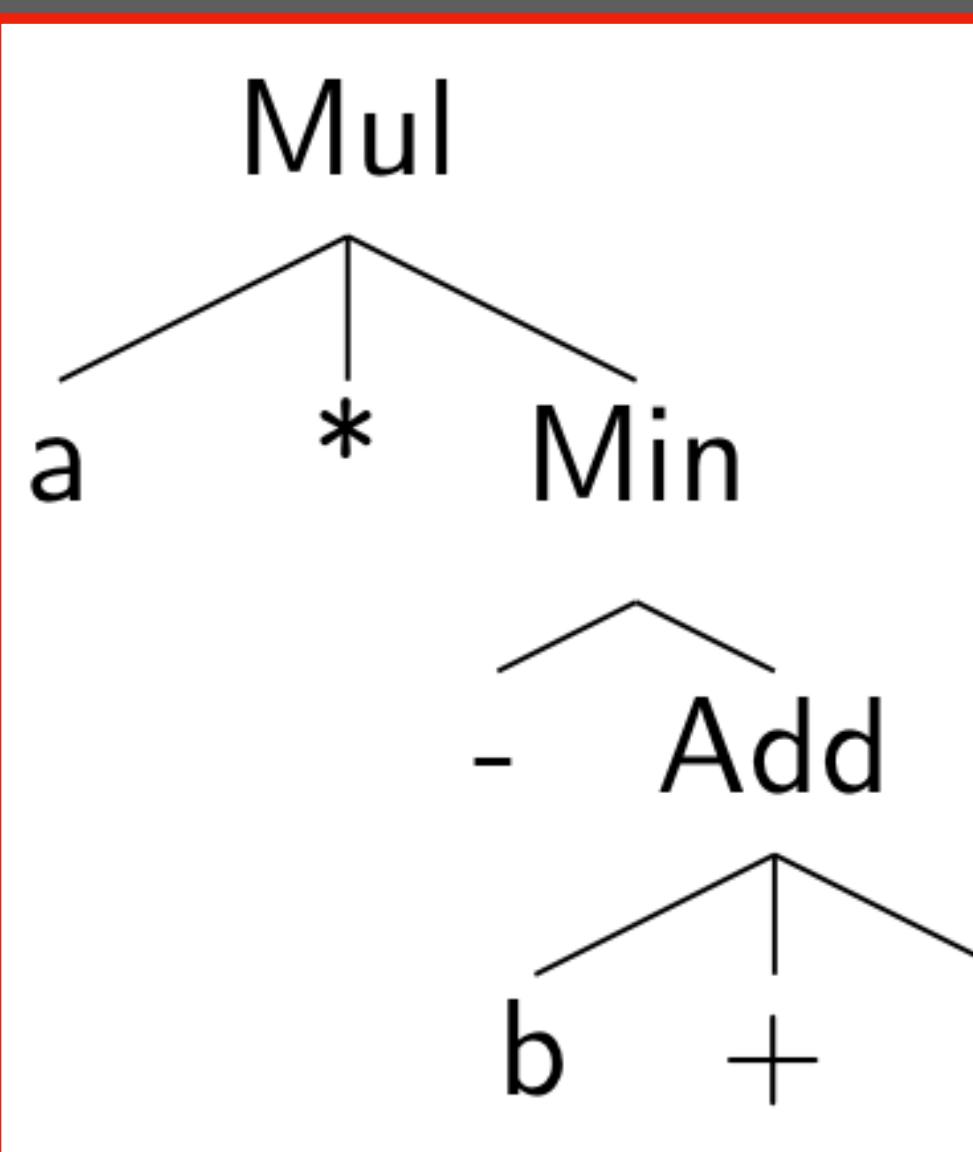


Safe for High Priority Prefix Operators

Conflict Patterns



Trees



Unsafe for Low Priority Prefix Operators [SDF2]

Conflict
Patterns

$$\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}$$

```
graph TD; Add --> Lam; Add --> Plus["+"]; Add --> E1[E]; Lam --> Lambda["λ"]; Lam --> ID["ID"]; Lam --> Dot["."]; E1 --> E2[E];
```

Trees

Unsafe for Low Priority Prefix Operators [SDF2]

Conflict
Patterns

$$\frac{E.Add > E.Lam}{\text{Add}}$$

An abstract syntax tree for the Add pattern. The root node is labeled "Add". It has three children: "Lam", "+", and "E". The "Lam" node has one child: " $\lambda \text{ ID . } E$ ". This tree represents the expression $(\lambda \text{ ID . } E) + \text{Lam}$.

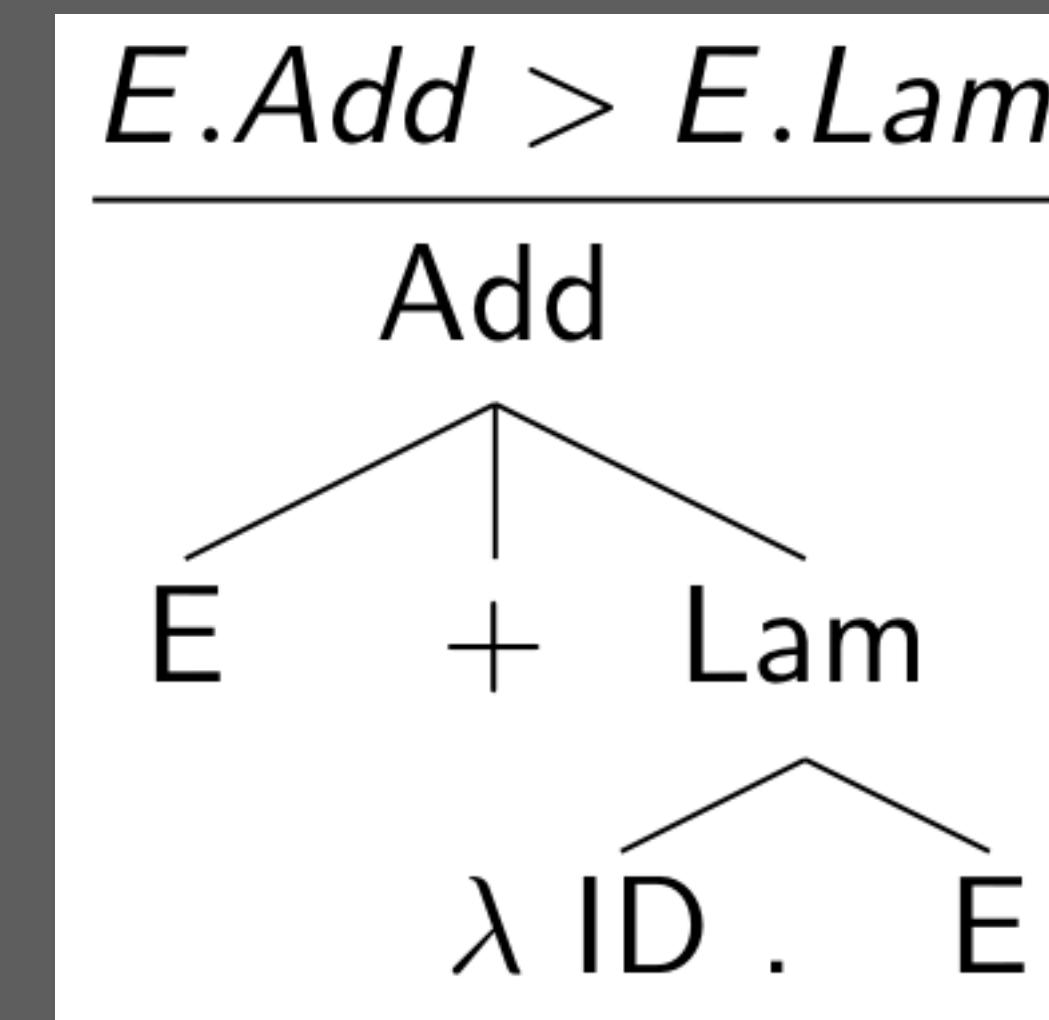
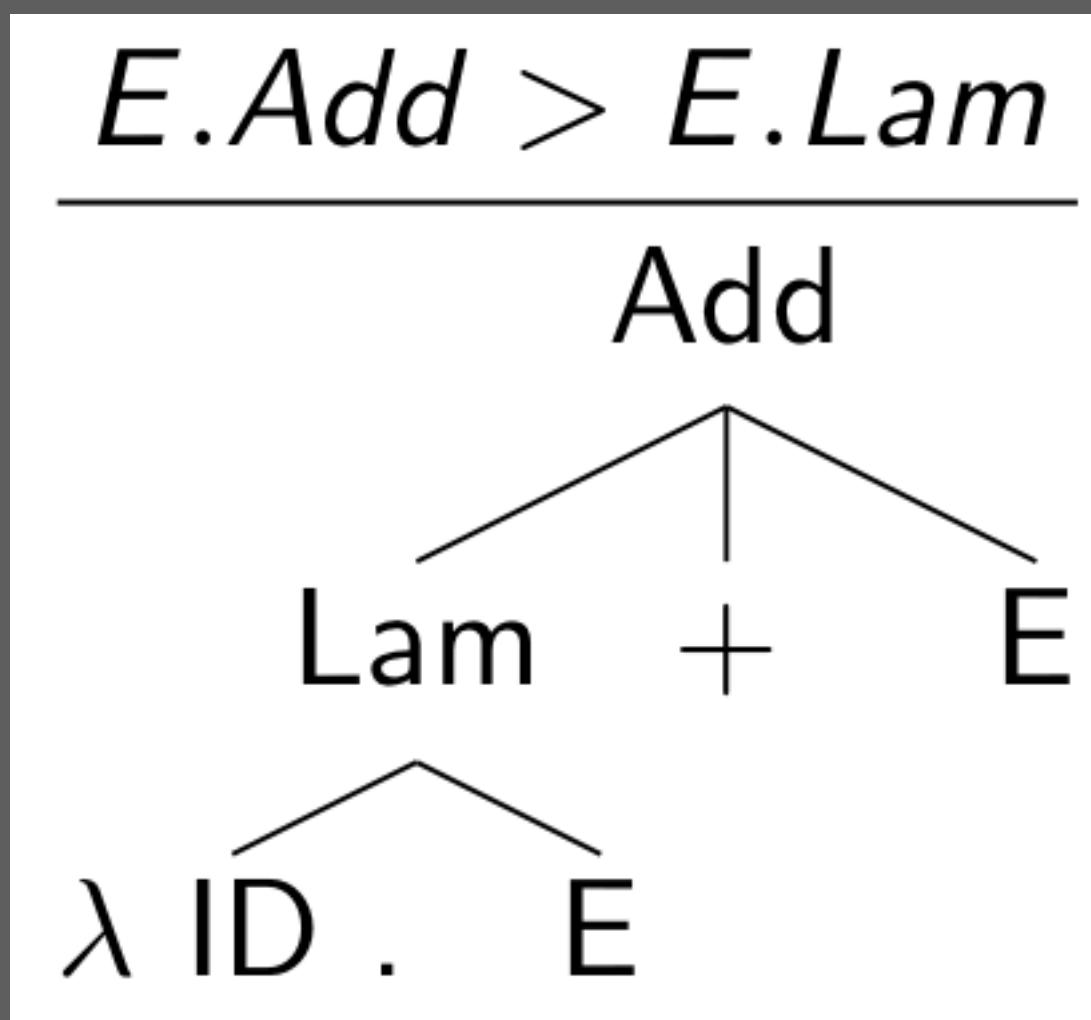
$$\frac{E.Add > E.Lam}{\text{Add}}$$

An abstract syntax tree for the Add pattern. The root node is labeled "Add". It has three children: "E", "+", and "Lam". The "Lam" node has one child: " $\lambda \text{ ID . } E$ ". This tree represents the expression $E + (\lambda \text{ ID . } E)$.

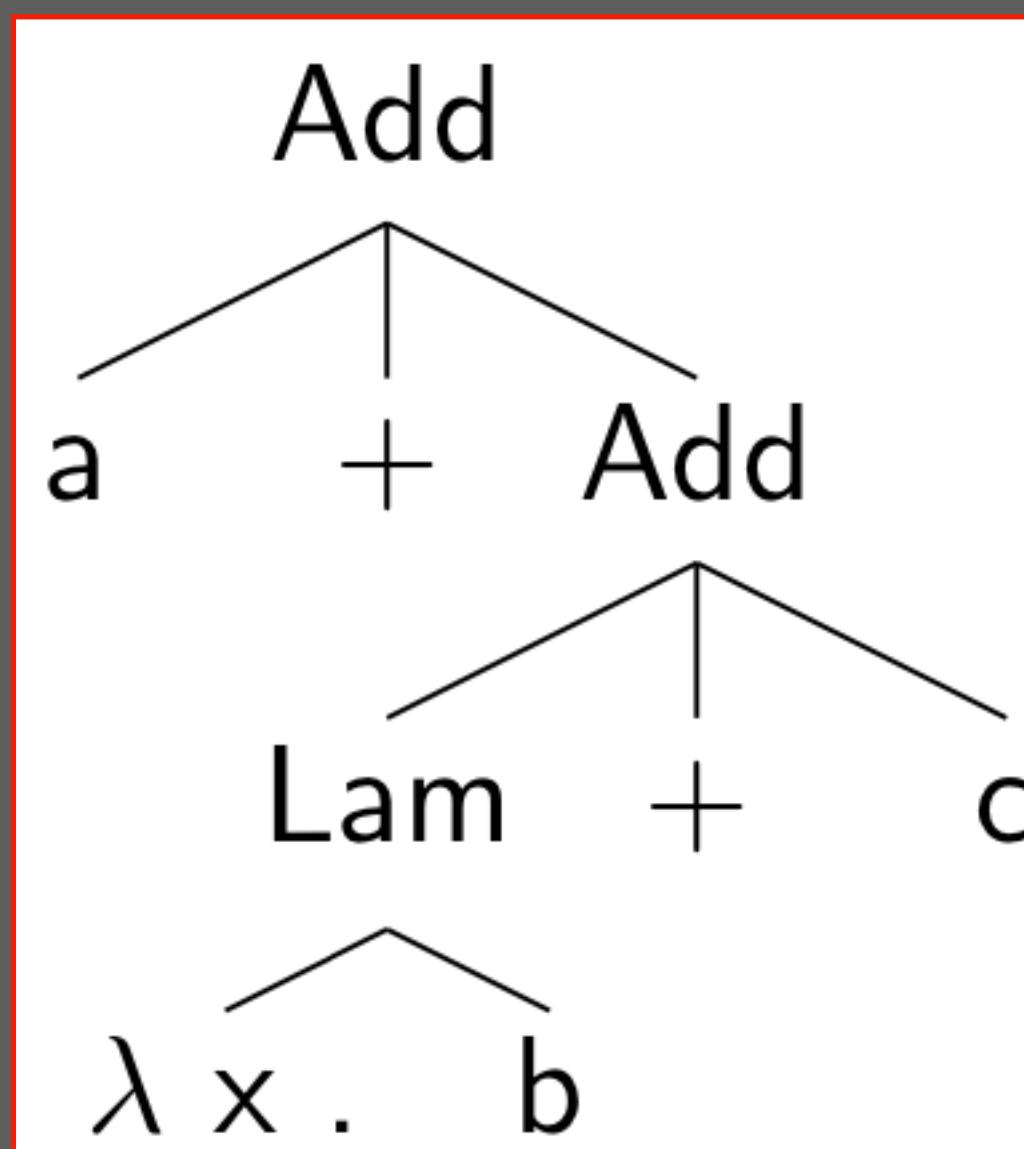
Trees

Unsafe for Low Priority Prefix Operators [SDF2]

Conflict Patterns

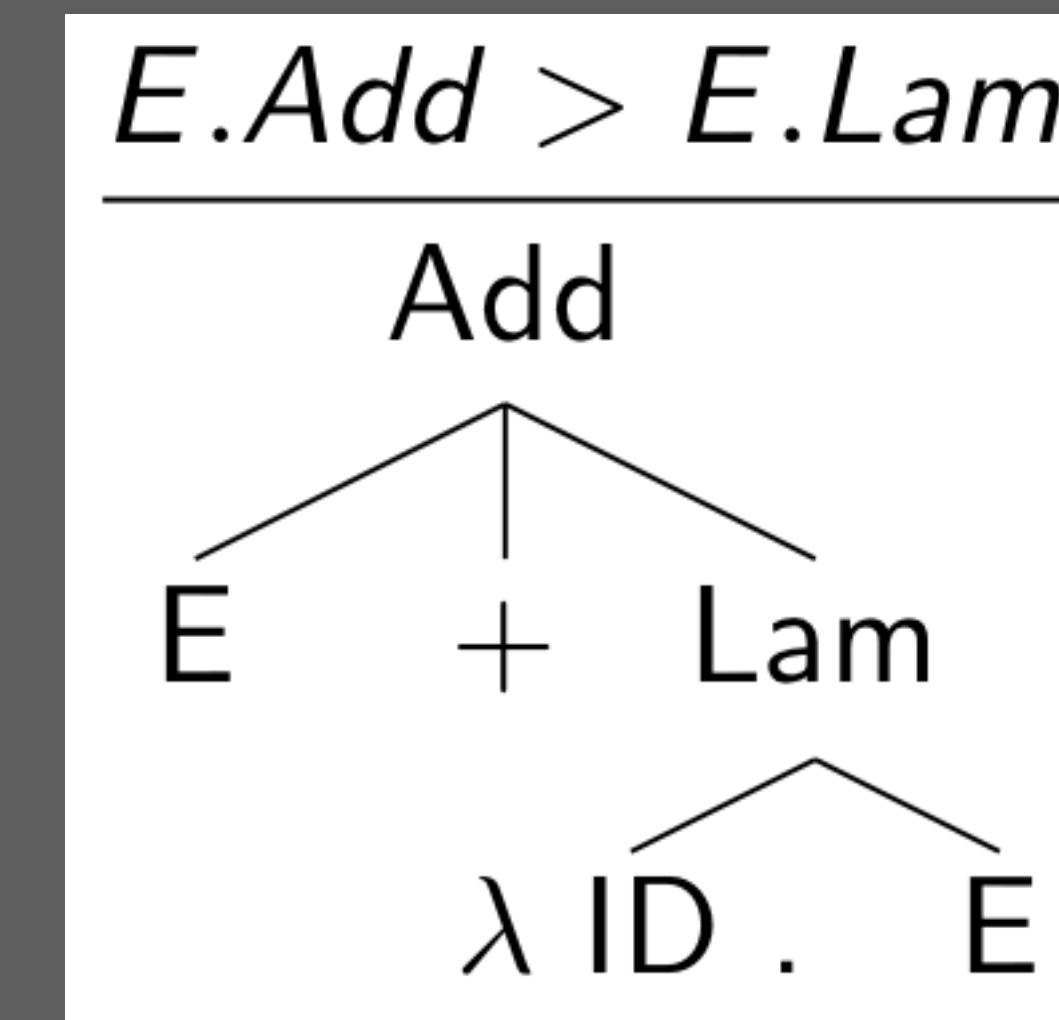
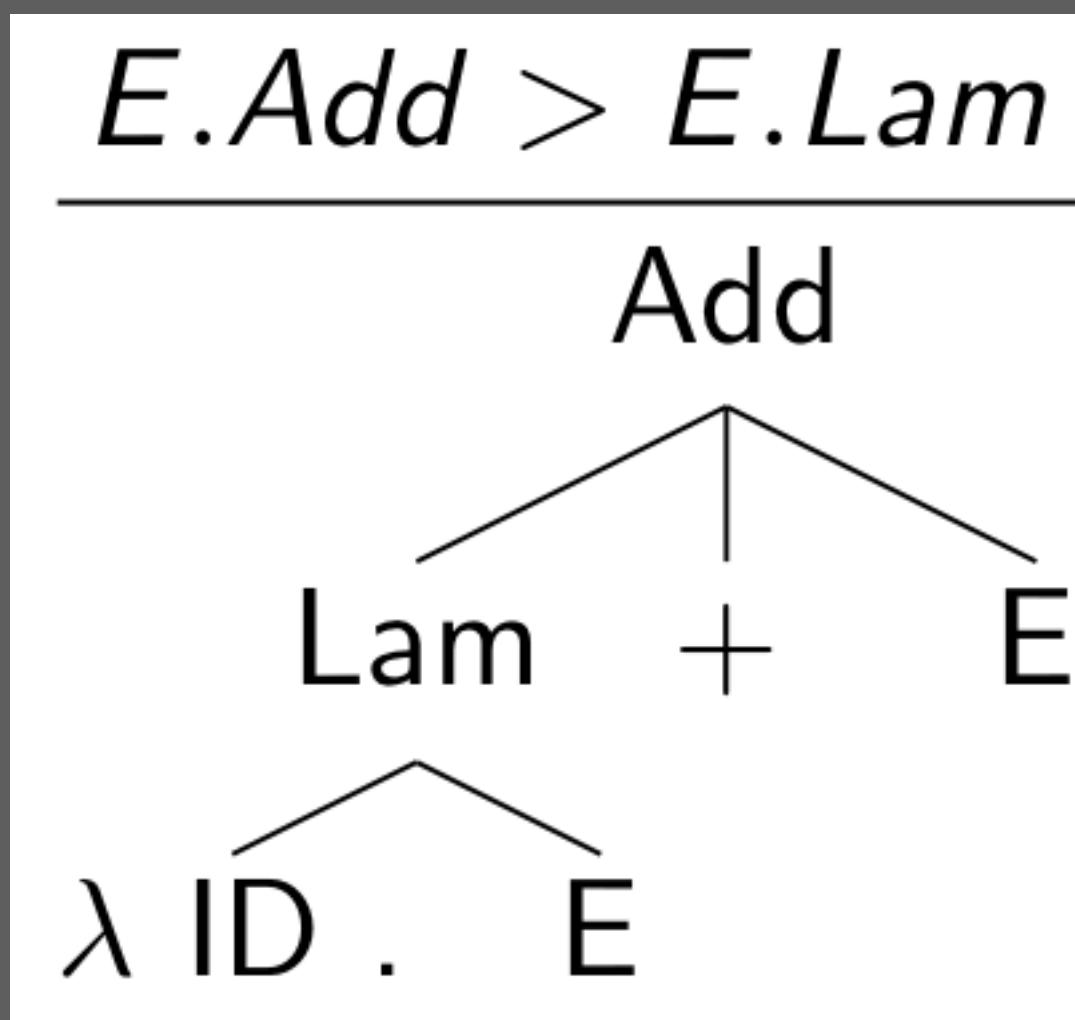


Trees

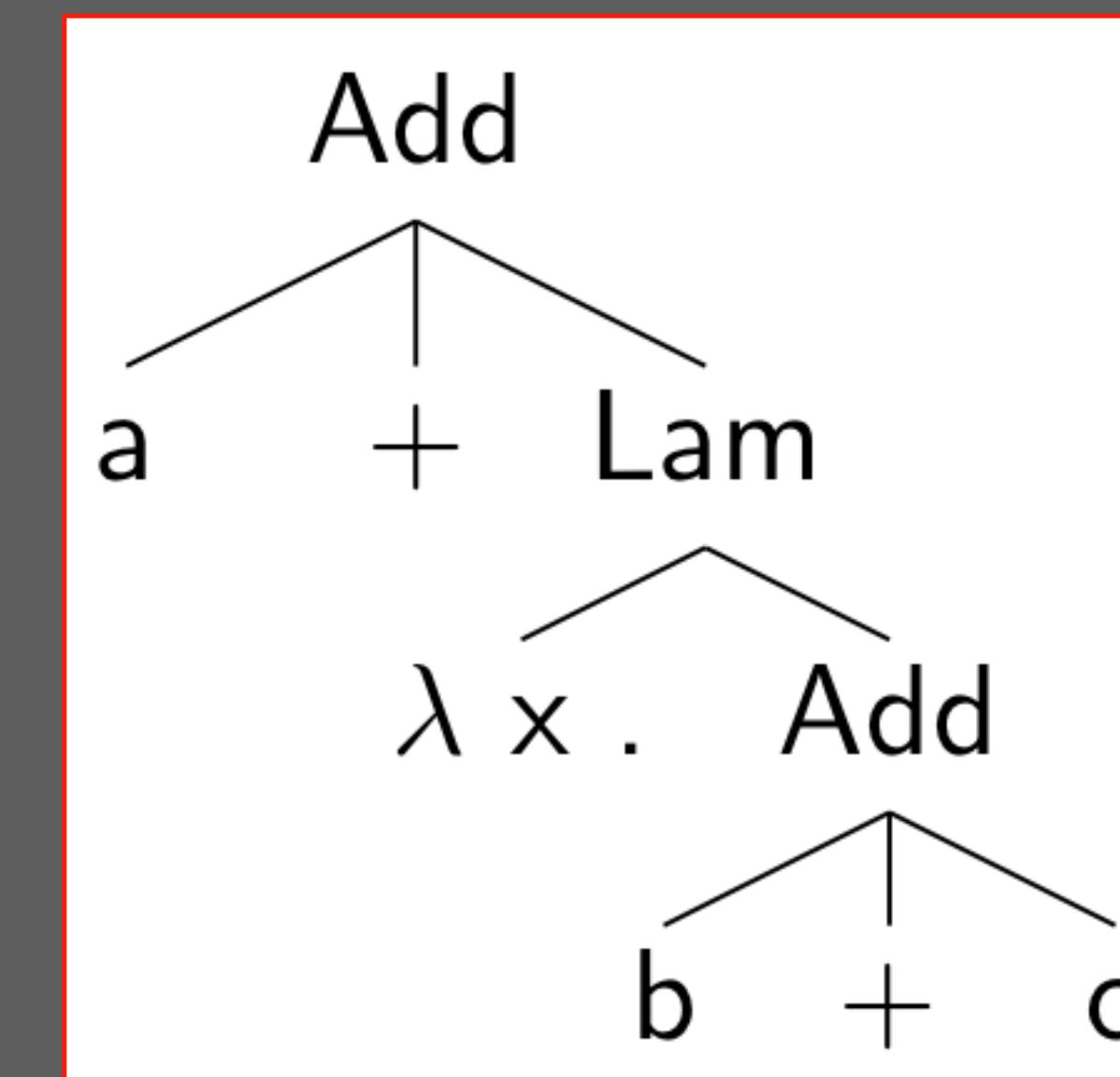
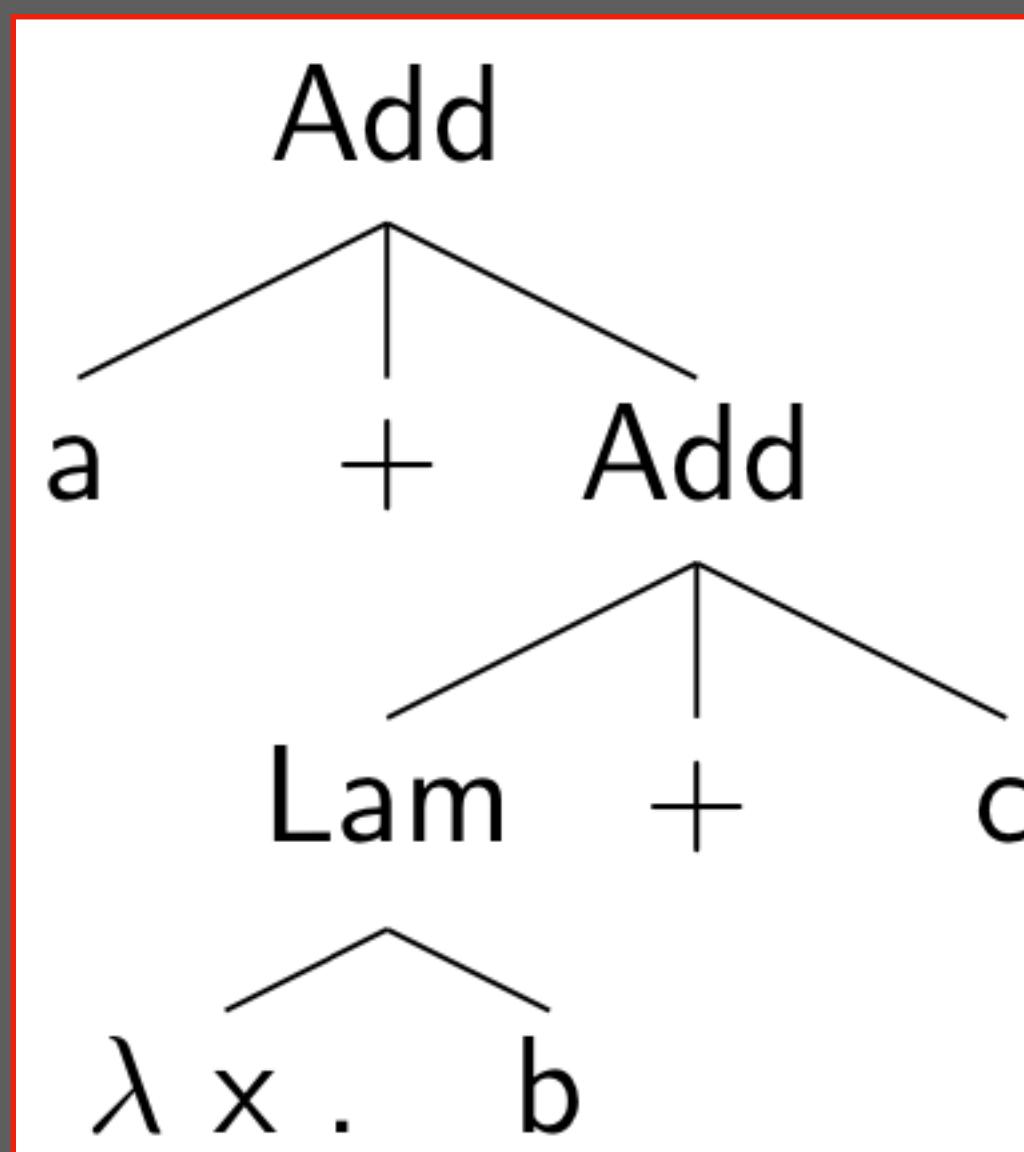


Unsafe for Low Priority Prefix Operators [SDF2]

Conflict Patterns

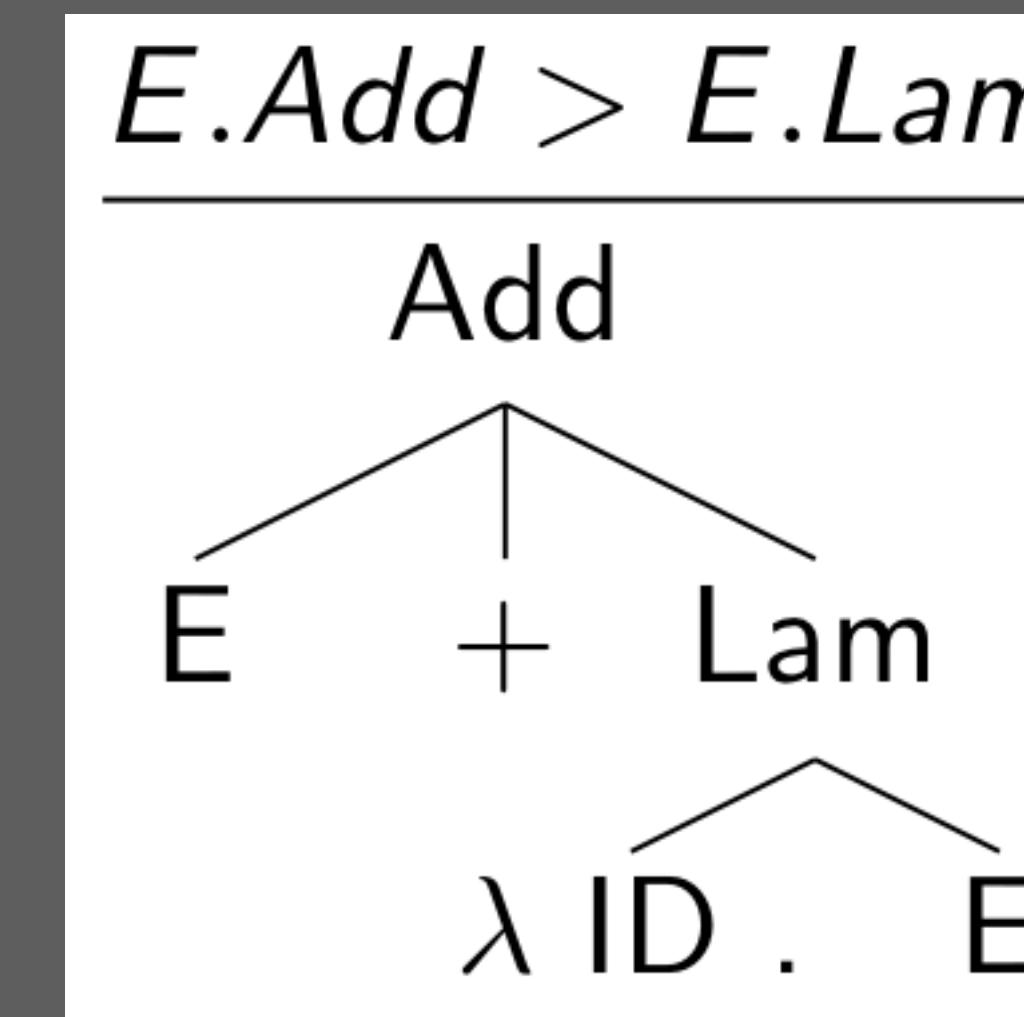
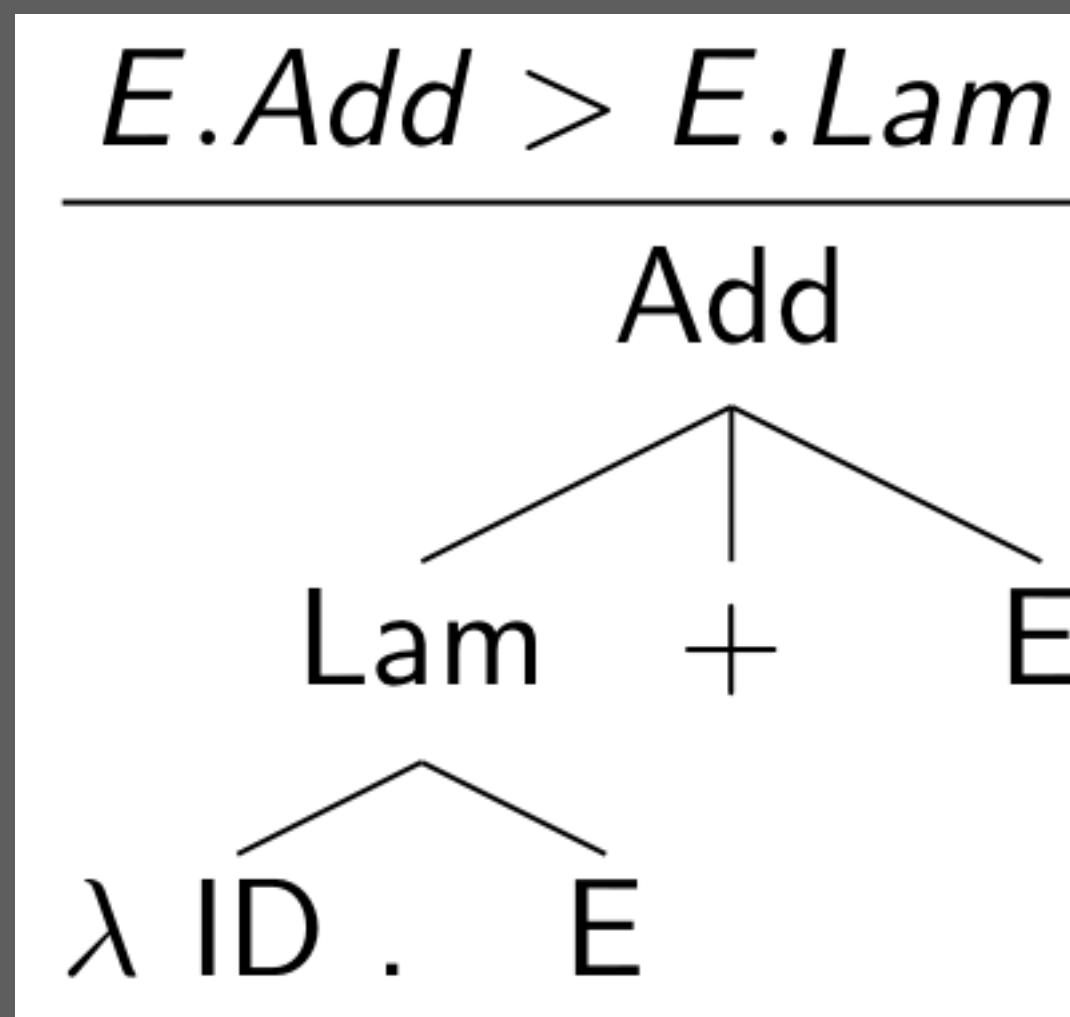


Trees

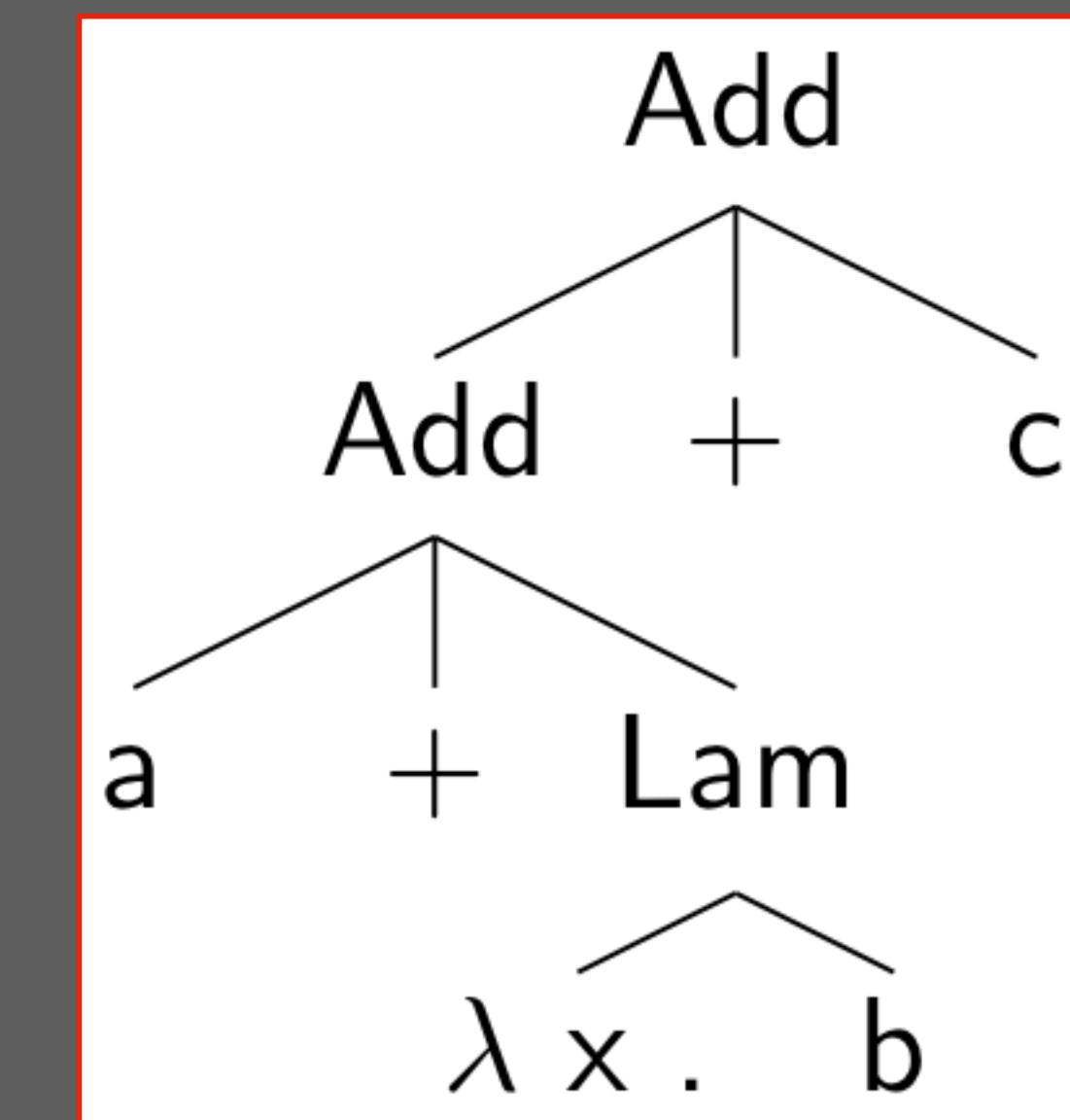
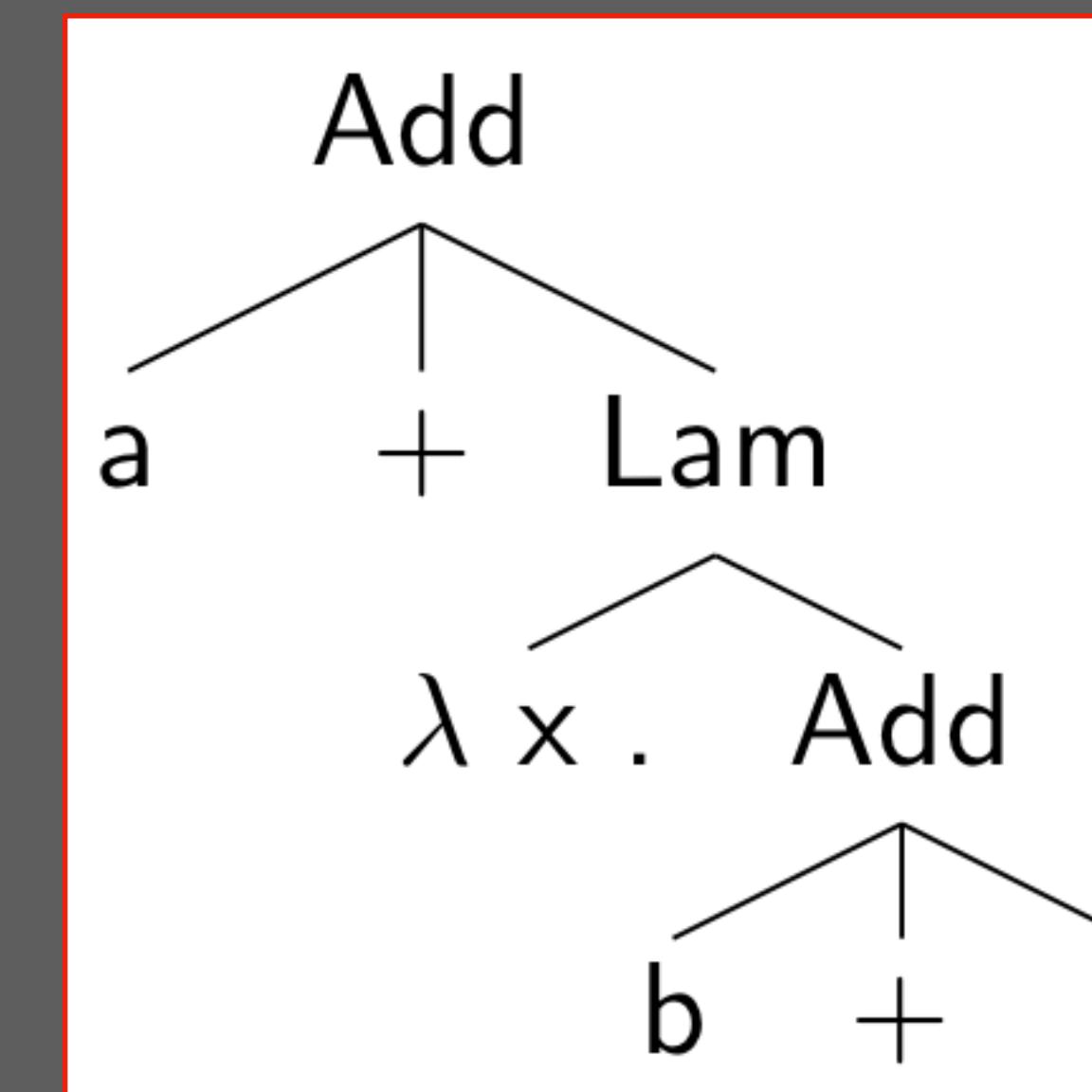
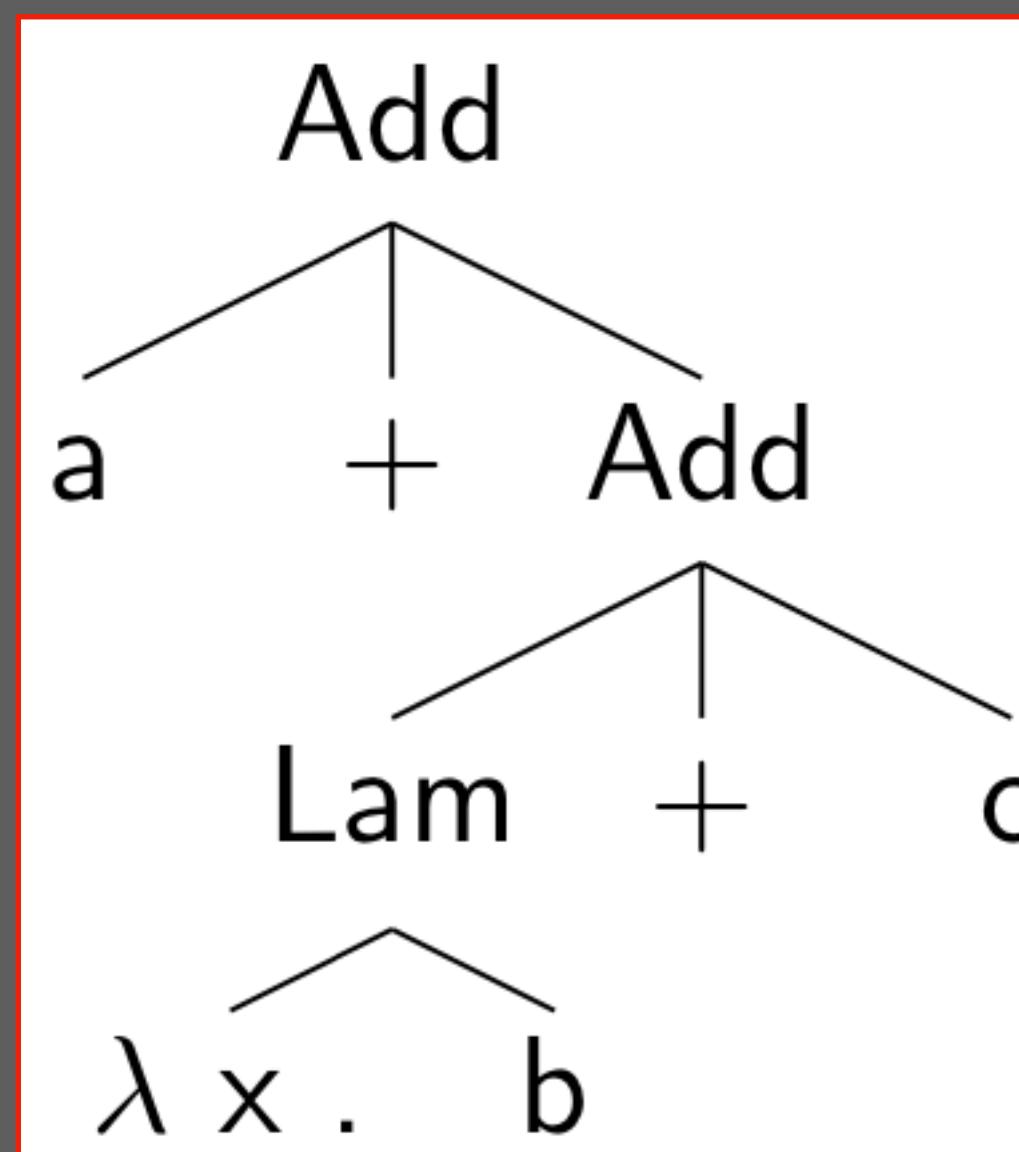


Unsafe for Low Priority Prefix Operators [SDF2]

Conflict Patterns



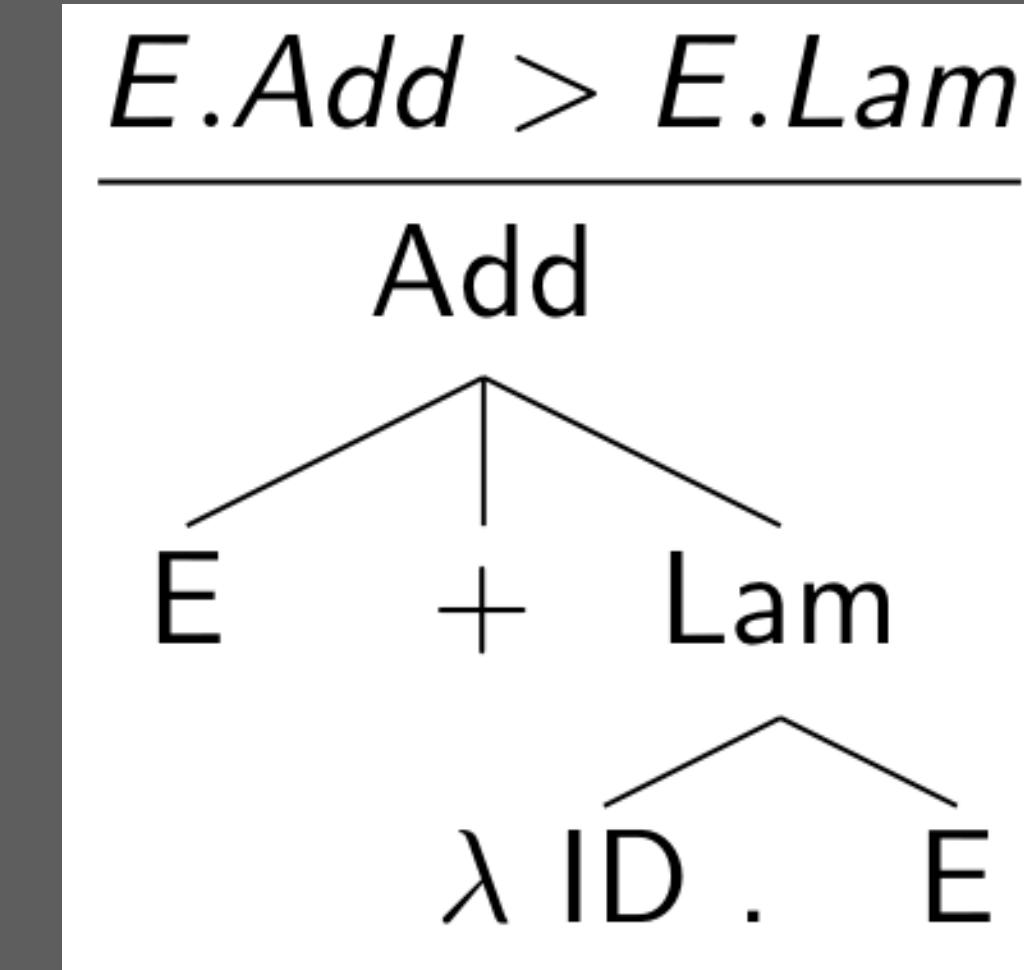
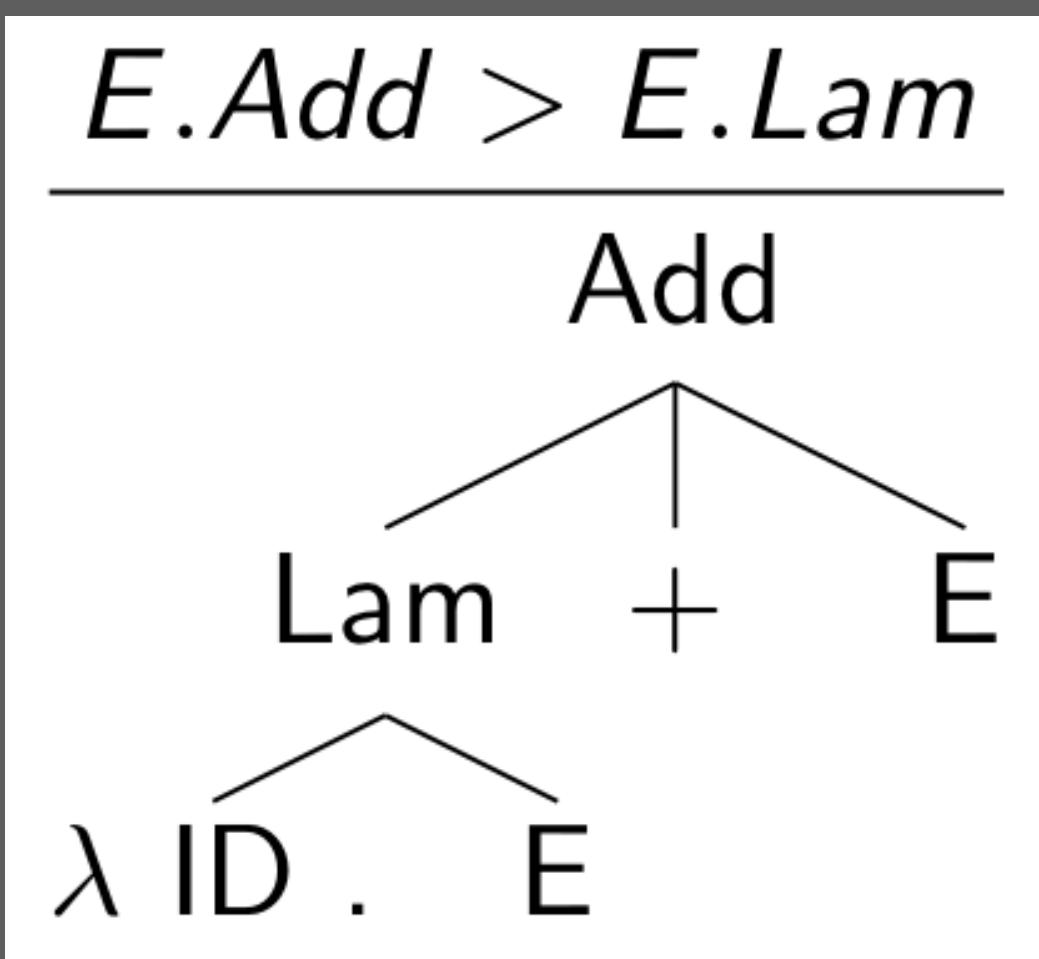
Trees



Safe Subtree Exclusion Rules [SDF3 (2019)]

Rules

Conflict Patterns



Safe Subtree Exclusion Rules [SDF3 (2019)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --> C2; C1 --> beta; C2 --> alpha; C2 --> A;
```

Right Recursive in
Left Recursive Position

Conflict
Patterns

$$\frac{E.Add > E.Lam}{Add}$$

```
graph TD; Add --> Lam; Add --> plus; Add --> E; E --> lambdaIDDotE;
```

$$\frac{E.Add > E.Lam}{Add}$$

```
graph TD; Add --> E; Add --> plus; Add --> Lam; E --> lambdaIDDotE;
```

Safe Subtree Exclusion Rules [SDF3 (2019)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --> C2; C1 --> beta; C2 --> alpha; C2 --> A;
```

Right Recursive in
Left Recursive Position

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --> alpha; C1 --> C2; C2 --> A; C2 --> beta;
```

Left Recursive in
Right Recursive Position

Conflict
Patterns

$$\frac{E.Add > E.Lam}{Add}$$

```
graph TD; Add --> Lam; Add --> plus; Lam --> lambda; Lam --> ID; Lam --> dot; lambda --> E;
```

$$\frac{E.Add > E.Lam}{Add}$$

```
graph TD; Add --> E; Add --> plus; Add --> Lam; Lam --> lambda; Lam --> ID; Lam --> dot; lambda --> E;
```

Safe Subtree Exclusion Rules [SDF3 (2019)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --> C2; C1 --> beta; C2 --> alpha; C2 --> A;
```

Right Recursive in
Left Recursive Position

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --> alpha; C1 --> C2; C2 --> A; C2 --> beta;
```

Left Recursive in
Right Recursive Position

Conflict
Patterns

$$\frac{E.Add > E.Lam}{Add}$$

```
graph TD; Add --> Lam; Add --> plus; Lam --> lambda; Lam --> ID; Lam --> dot; Lam --> E;
```

conflict pattern:
\ right recursive

$$\frac{E.Add > E.Lam}{Add}$$

```
graph TD; Add --> E; Add --> plus; Add --> Lam; E --> lambda; E --> ID; E --> dot; E --> E;
```

Safe Subtree Exclusion Rules [SDF3 (2019)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --> C2; C1 --> beta; C2 --> alpha; C2 --> A;
```

Right Recursive in
Left Recursive Position

$$\frac{A.C_1 > A.C_2}{C_1}$$

```
graph TD; C1 --> alpha; C1 --> C2; C2 --> A; C2 --> beta;
```

Left Recursive in
Right Recursive Position

Conflict
Patterns

$$\frac{E.Add > E.Lam}{Add}$$

```
graph TD; Add --> Lam; Add --> plus; Add --> E; Lam --> lambda; Lam --> ID; Lam --> dot; Lam --> E;
```

conflict pattern:
\ right recursive

$$\frac{E.Add > E.Lam}{Add}$$

```
graph TD; Add --> E; Add --> plus; Add --> Lam; E -- red; plus -- red; Lam -- black; Lam --> lambda; Lam --> ID; Lam --> dot; Lam --> E;
```

not a conflict pattern:
\ not left recursive

Safe Subtree Exclusion Rules [SDF3 (2019)]

Rules

$$\frac{A.C_1 > A.C_2}{C_1}$$

```

graph TD
    C1[C1] --- C2[C2]
    C1 --- beta[β]
    C2 --- alpha[α]
    C2 --- A[A]
  
```

Right Recursive in
Left Recursive Position

$$\frac{A.C_1 > A.C_2}{C_1}$$

```

graph TD
    C1[C1] --- alpha[α]
    C1 --- C2[C2]
    C2 --- A[A]
    C2 --- beta[β]
  
```

Left Recursive in
Right Recursive Position

$$\frac{A.C_1 \text{ left } A.C_2}{C_1}$$

```

graph TD
    C1[C1] --- A[A]
    C1 --- alpha[α]
    A --- C2[C2]
    A --- beta[β]
    C2 --- A1[A]
    C2 --- A2[A]
  
```

Associativity

Conflict
Patterns

$$\frac{E.Add > E.Lam}{Add}$$

```

graph TD
    Add[Add] --- Lam[Lam]
    Add --- plus[+]
    Lam --- lambda[λ]
    Lam --- ID[ID]
    plus --- E[E]
    E --- E2[E]
  
```

conflict pattern:
\ right recursive

$$\frac{E.Add > E.Lam}{Add}$$

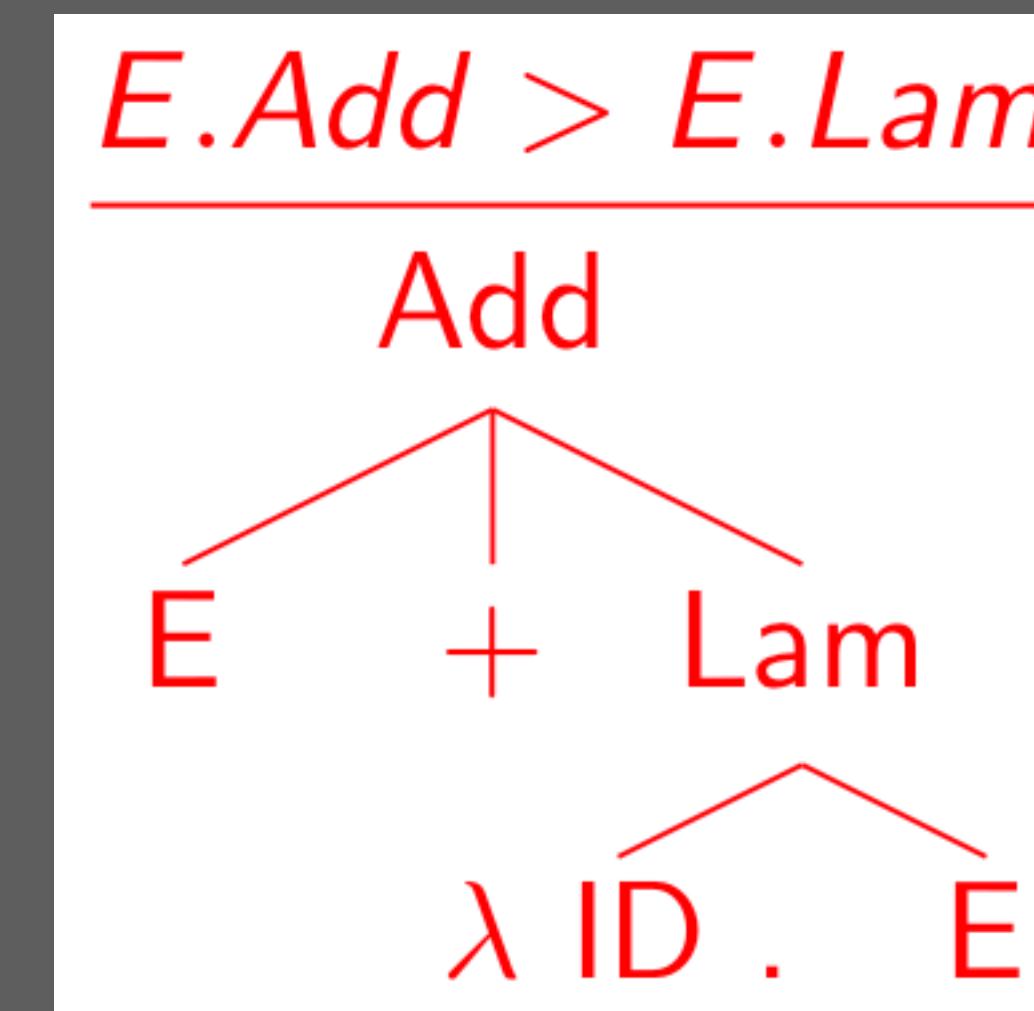
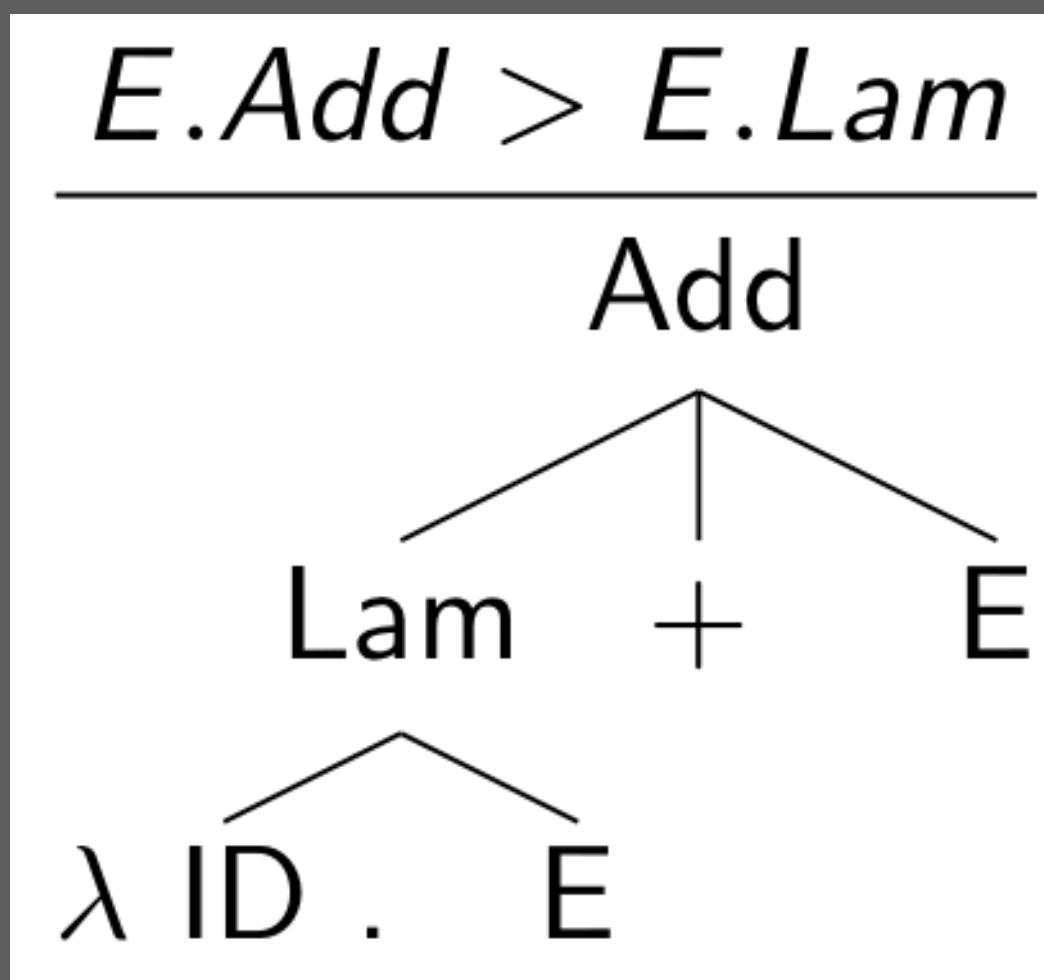
```

graph TD
    Add[Add] --- E[E]
    Add --- plus[+]
    Add --- Lam[Lam]
    E --- E2[E]
    plus --- E3[E]
    Lam --- lambda[λ]
    Lam --- ID[ID]
    E --- E4[E]
  
```

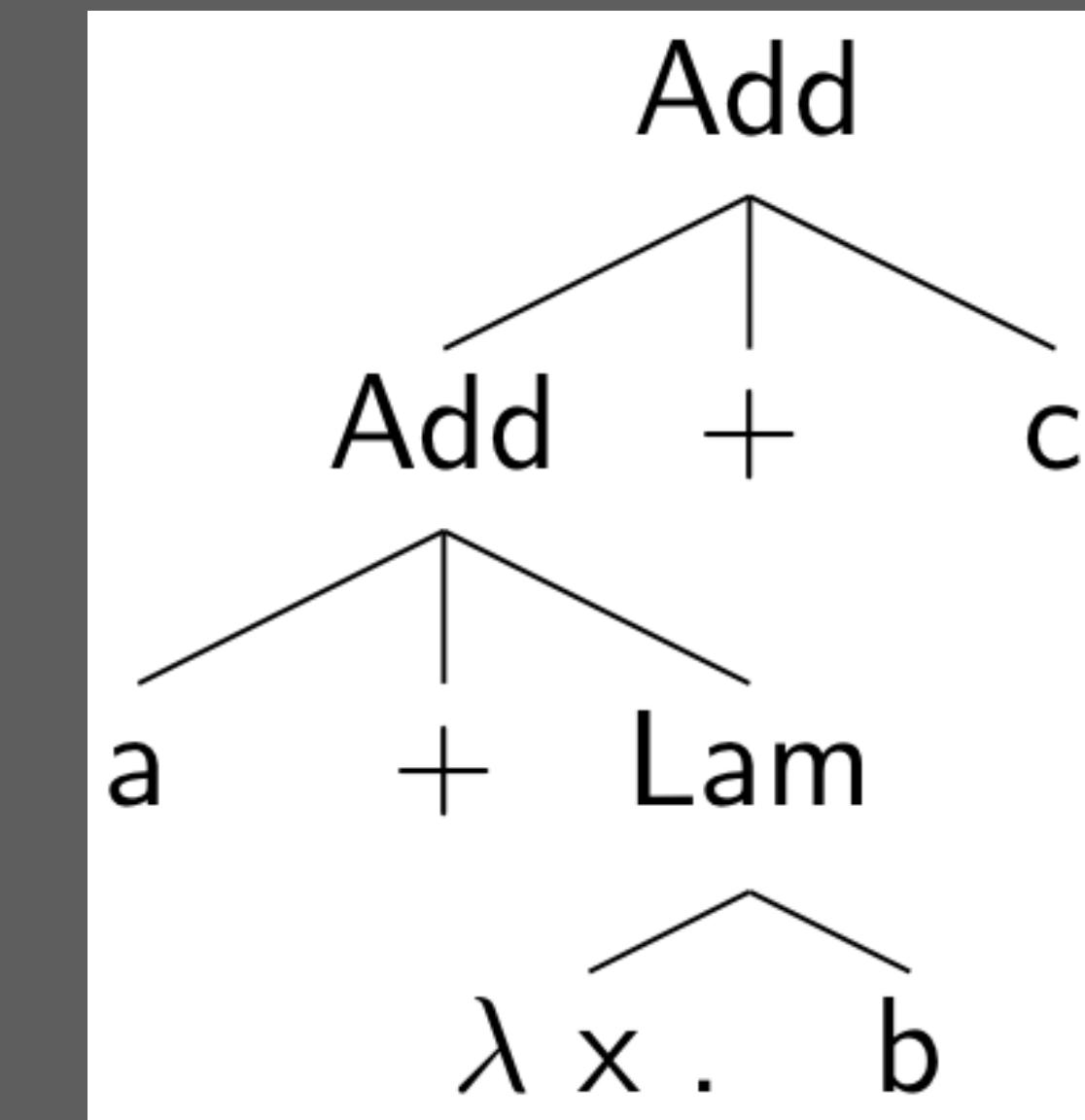
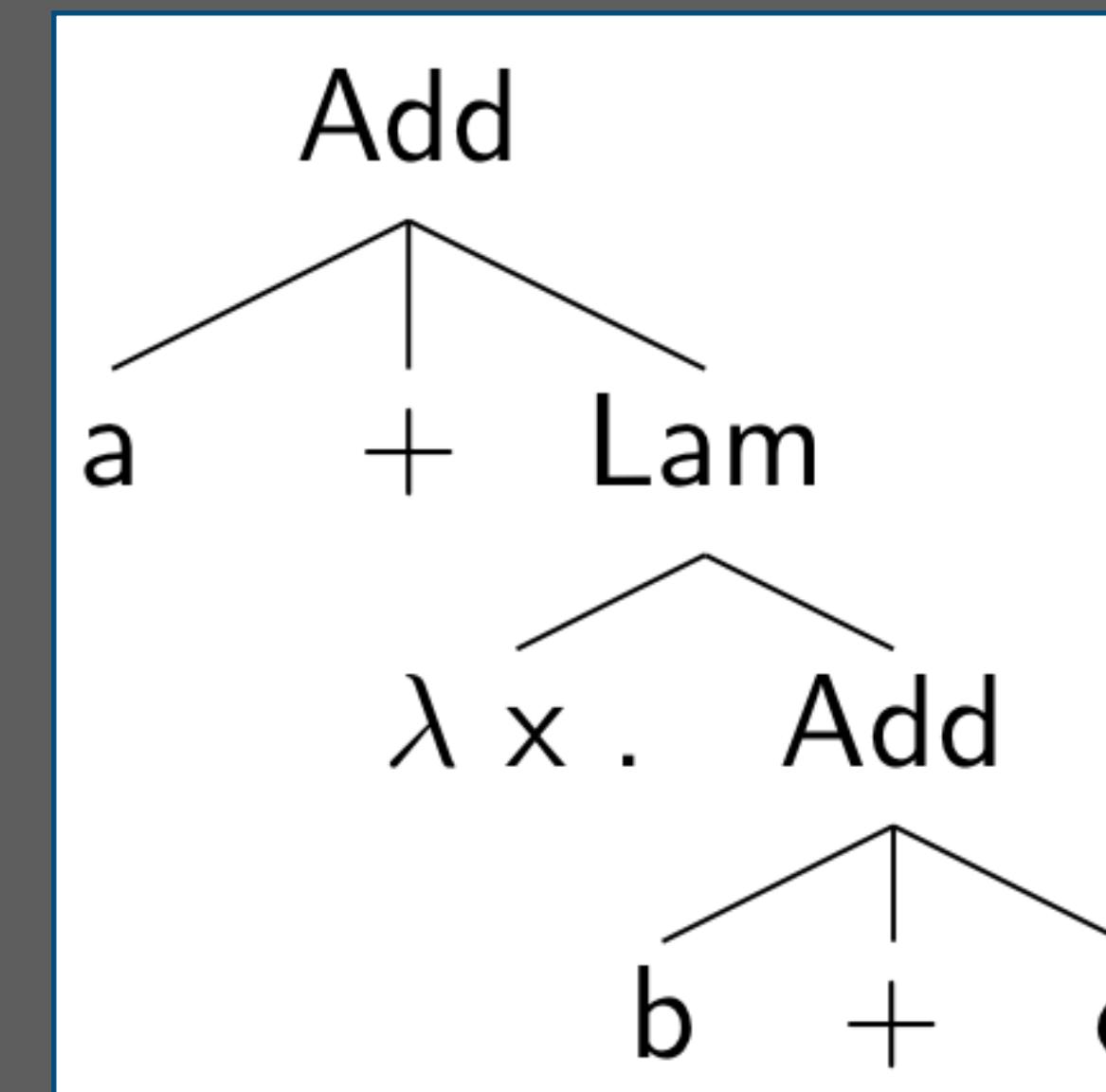
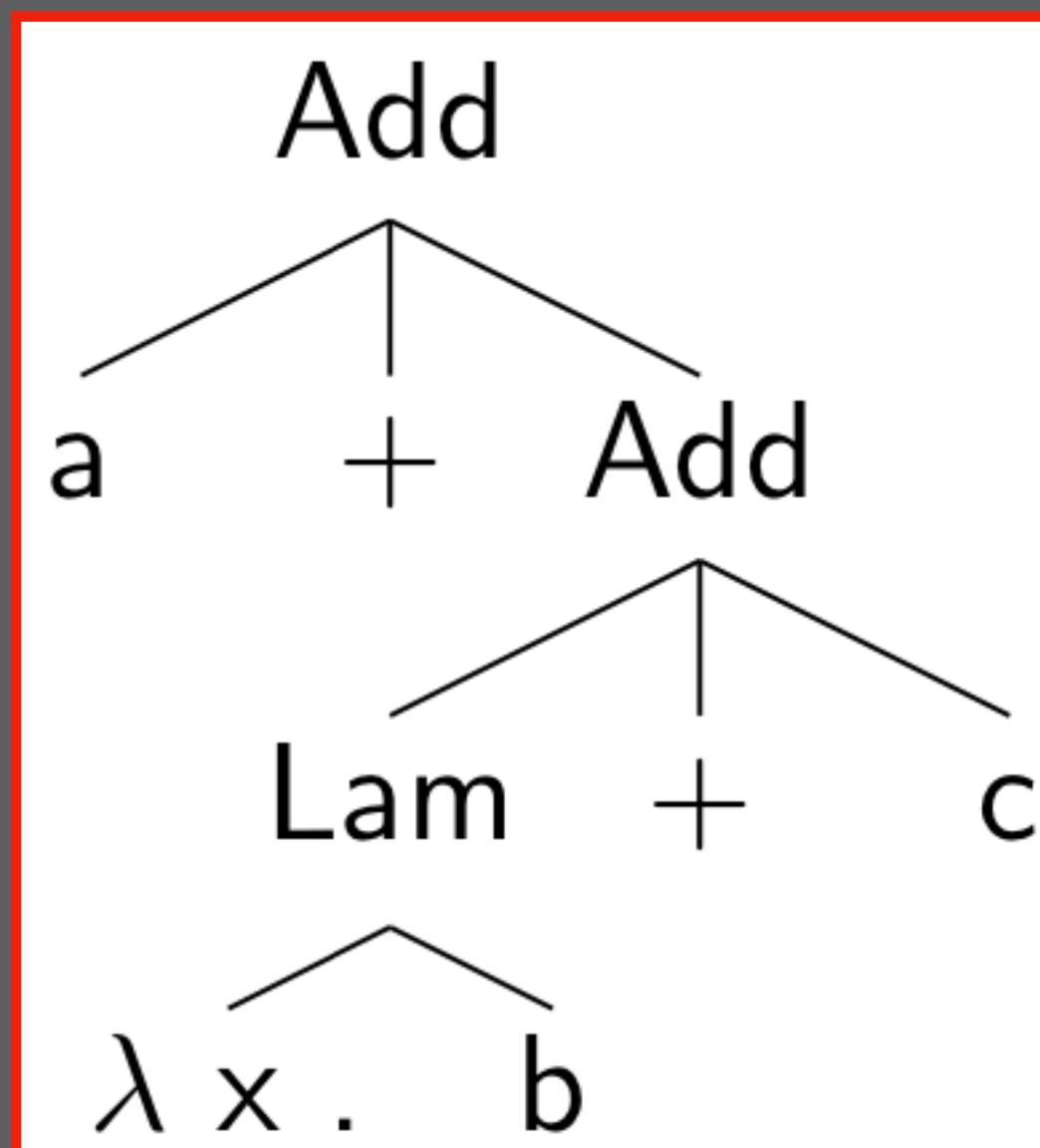
not a conflict pattern:
\ not left recursive

Shallow Interpretation: Safe for Low Priority Prefix Operators

Conflict Patterns

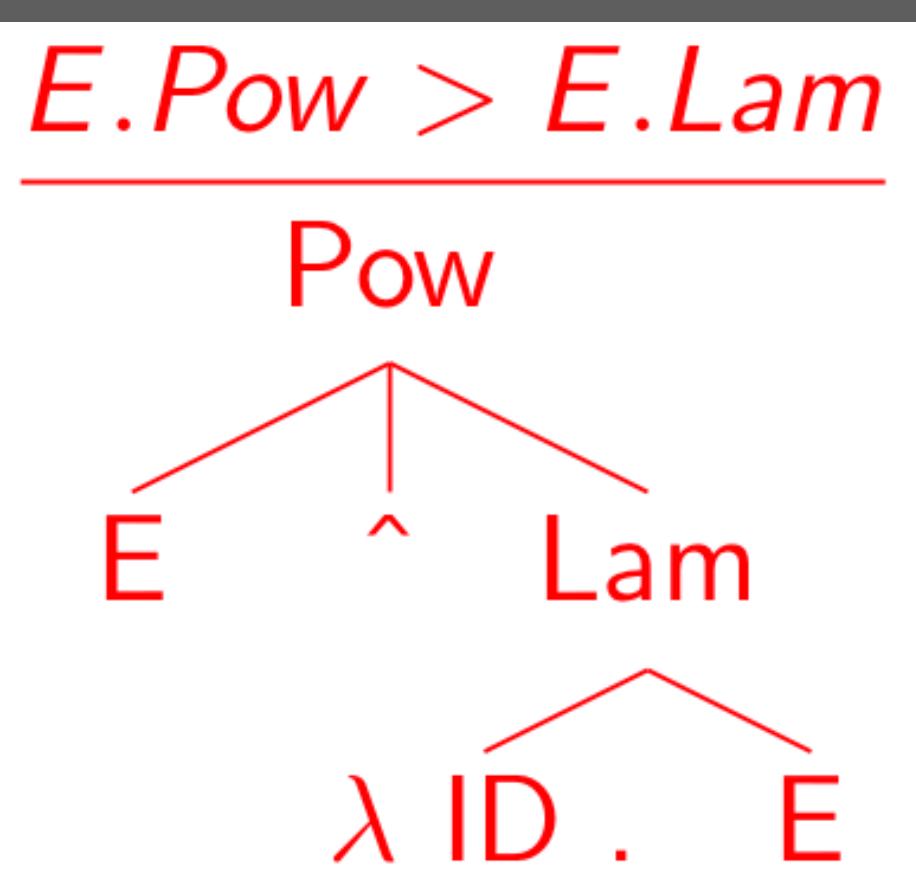
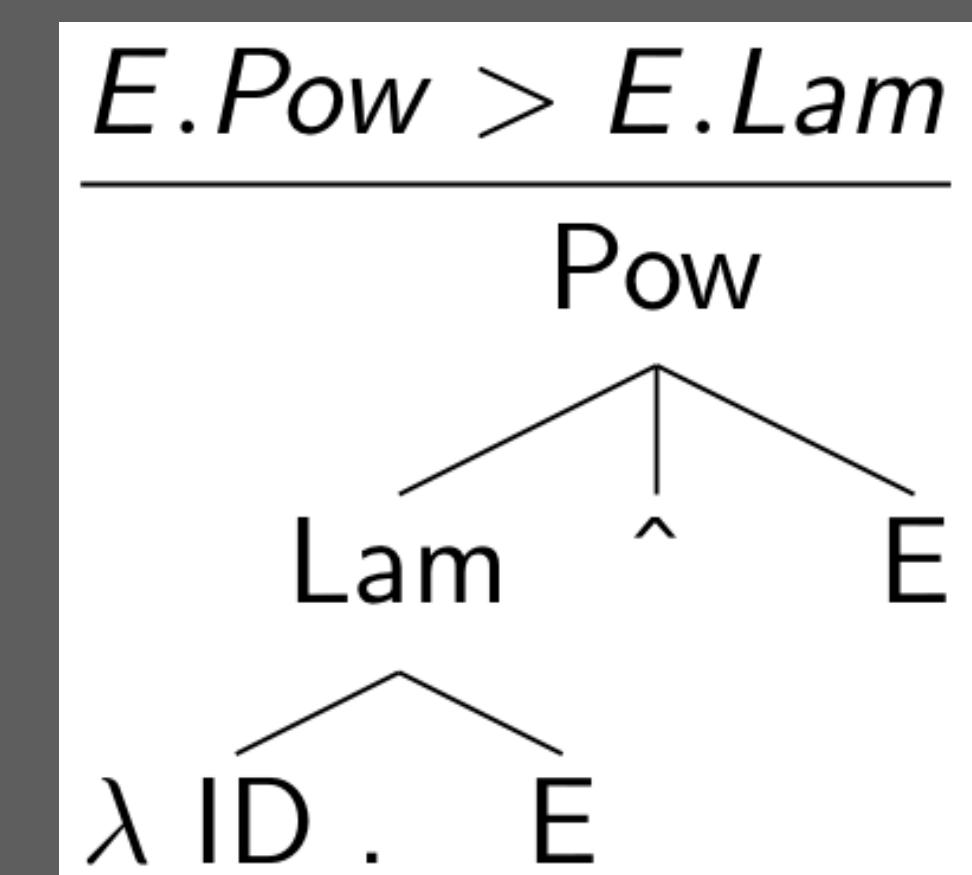
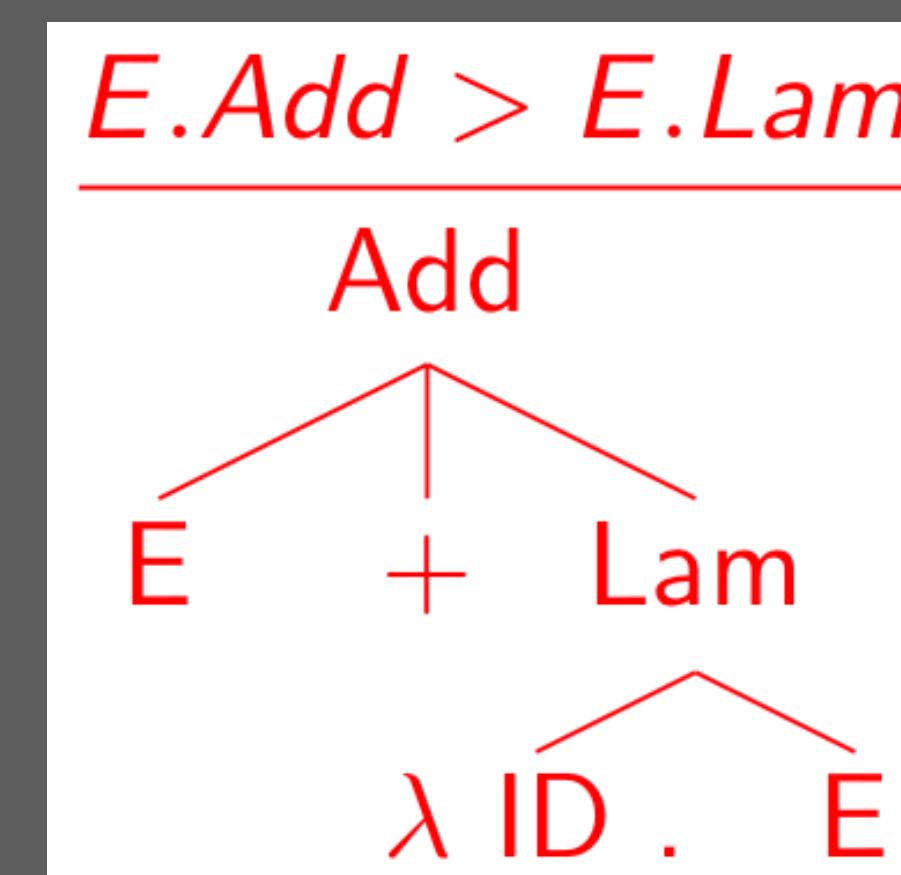
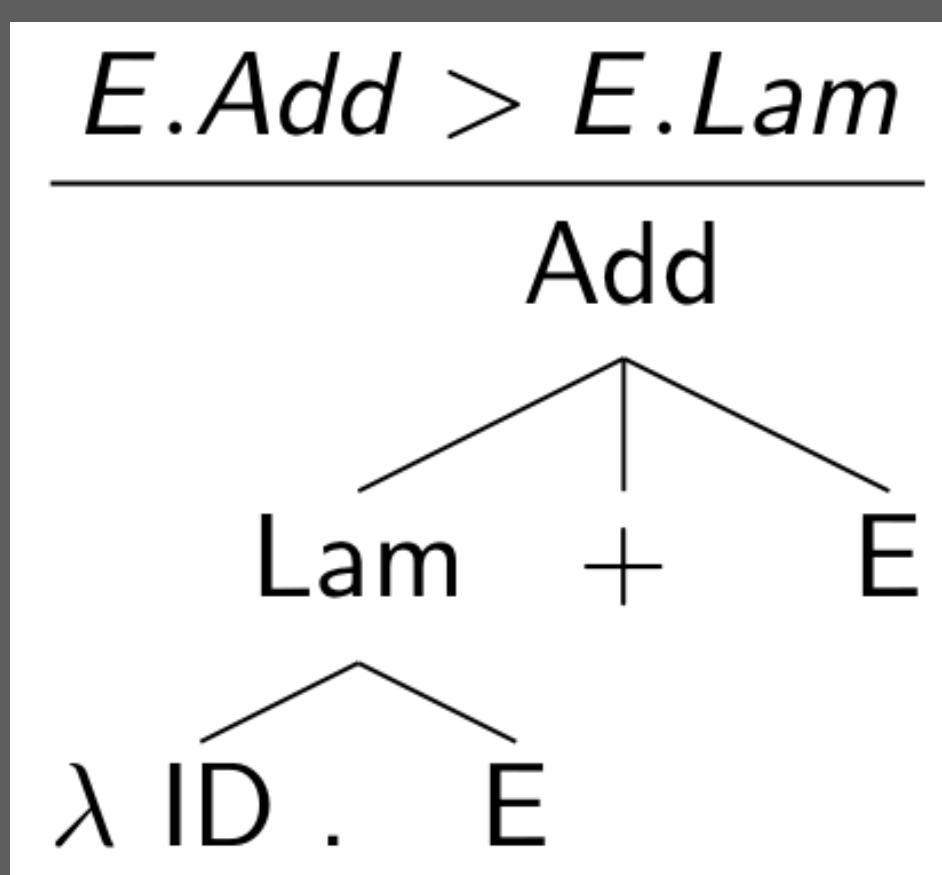


Trees



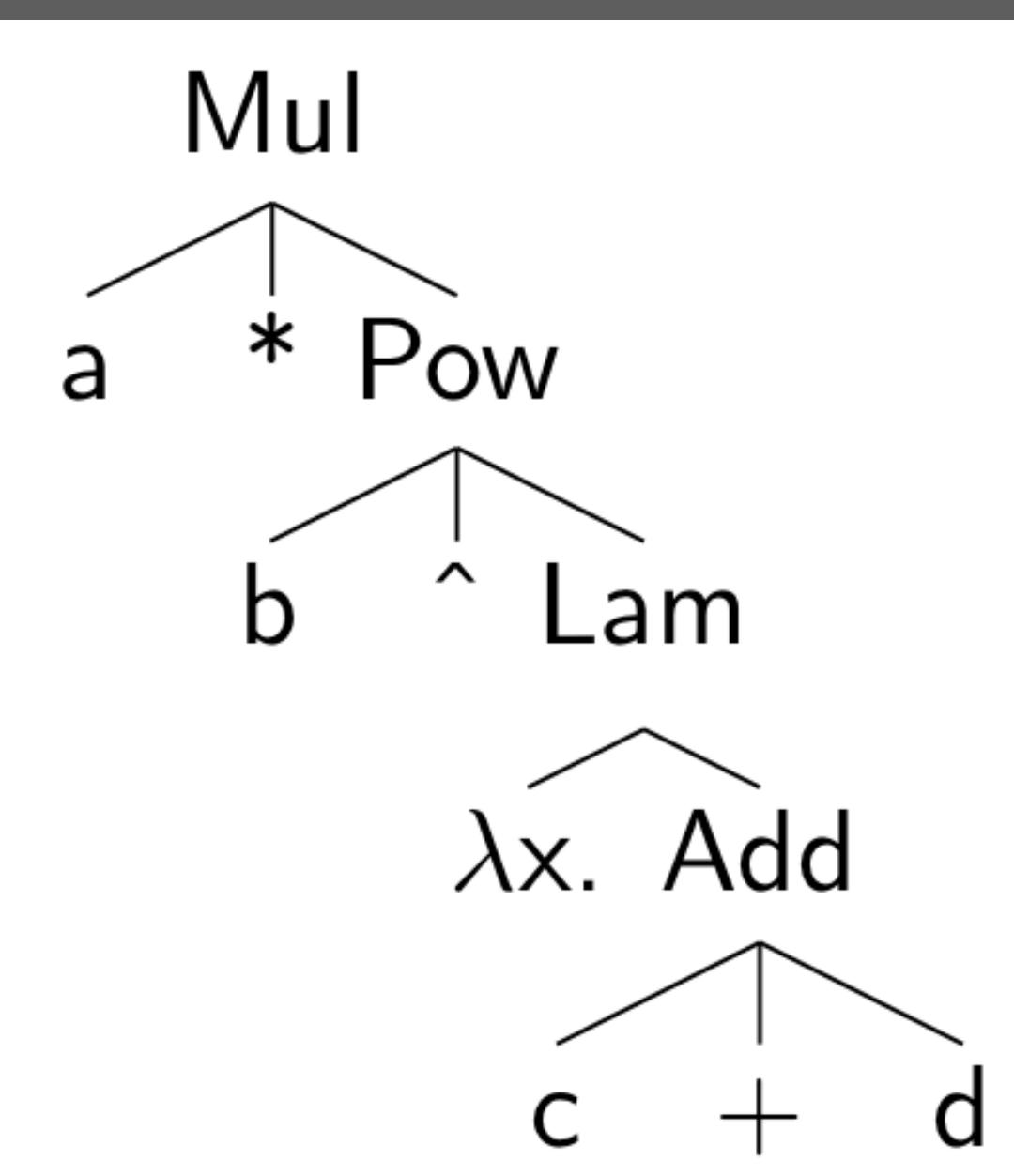
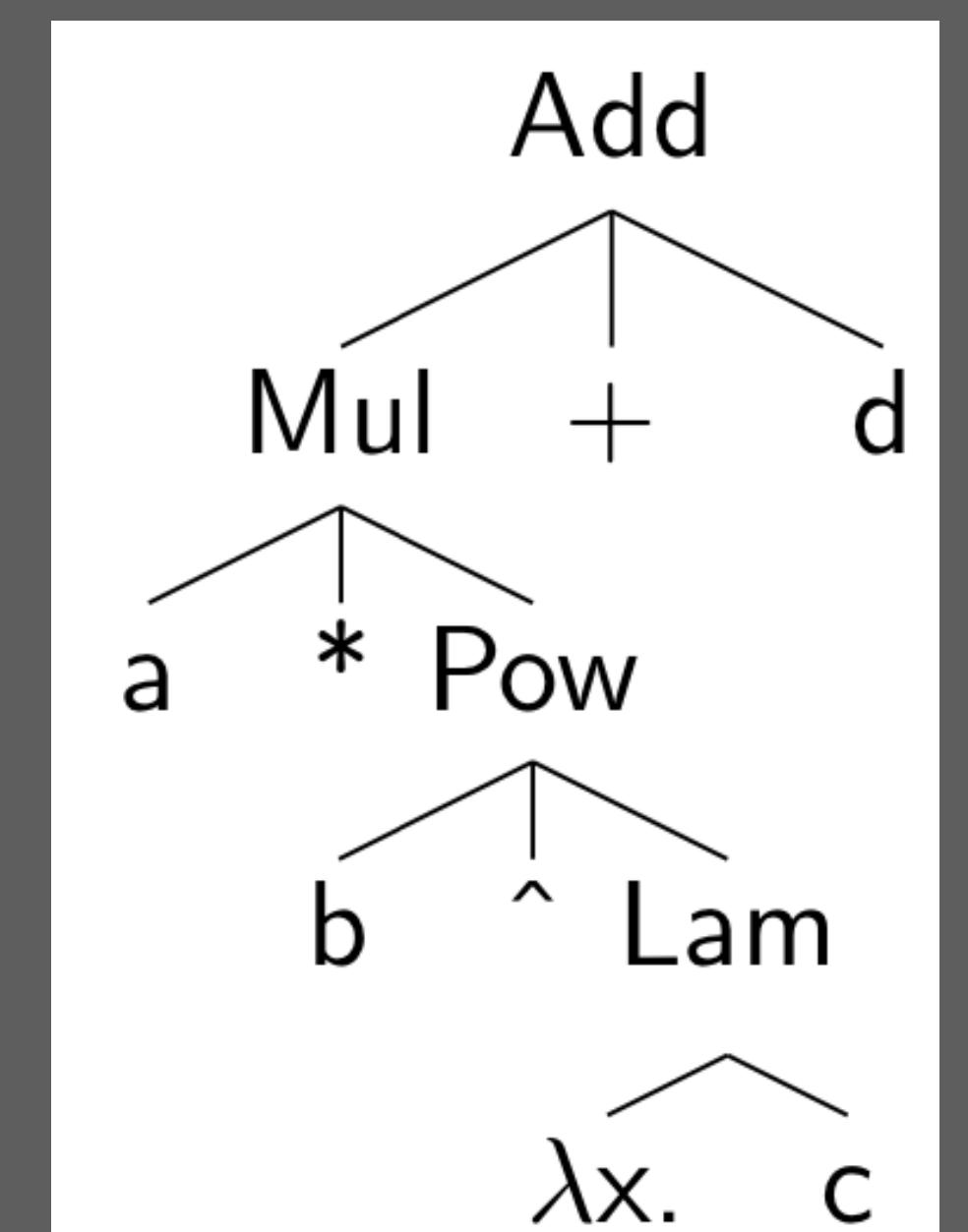
Shallow Interpretation: Incomplete for Low Priority Prefix Operators

Conflict Patterns

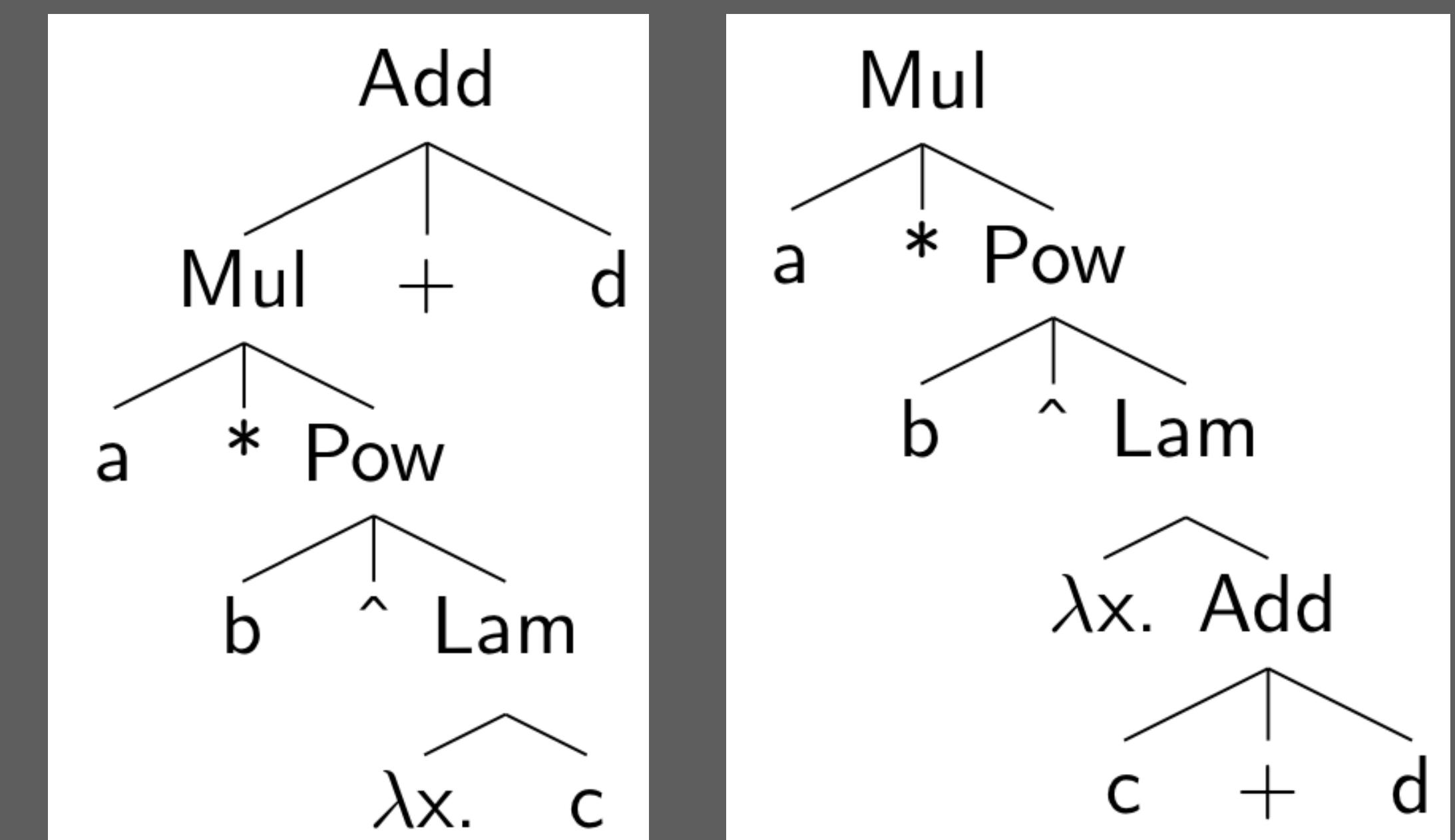
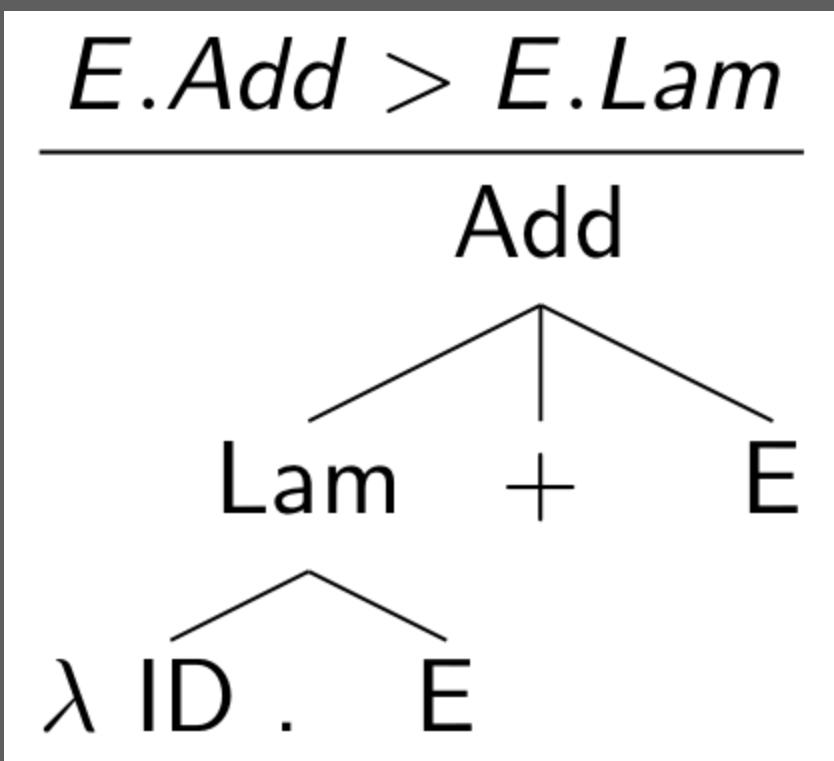


Trees

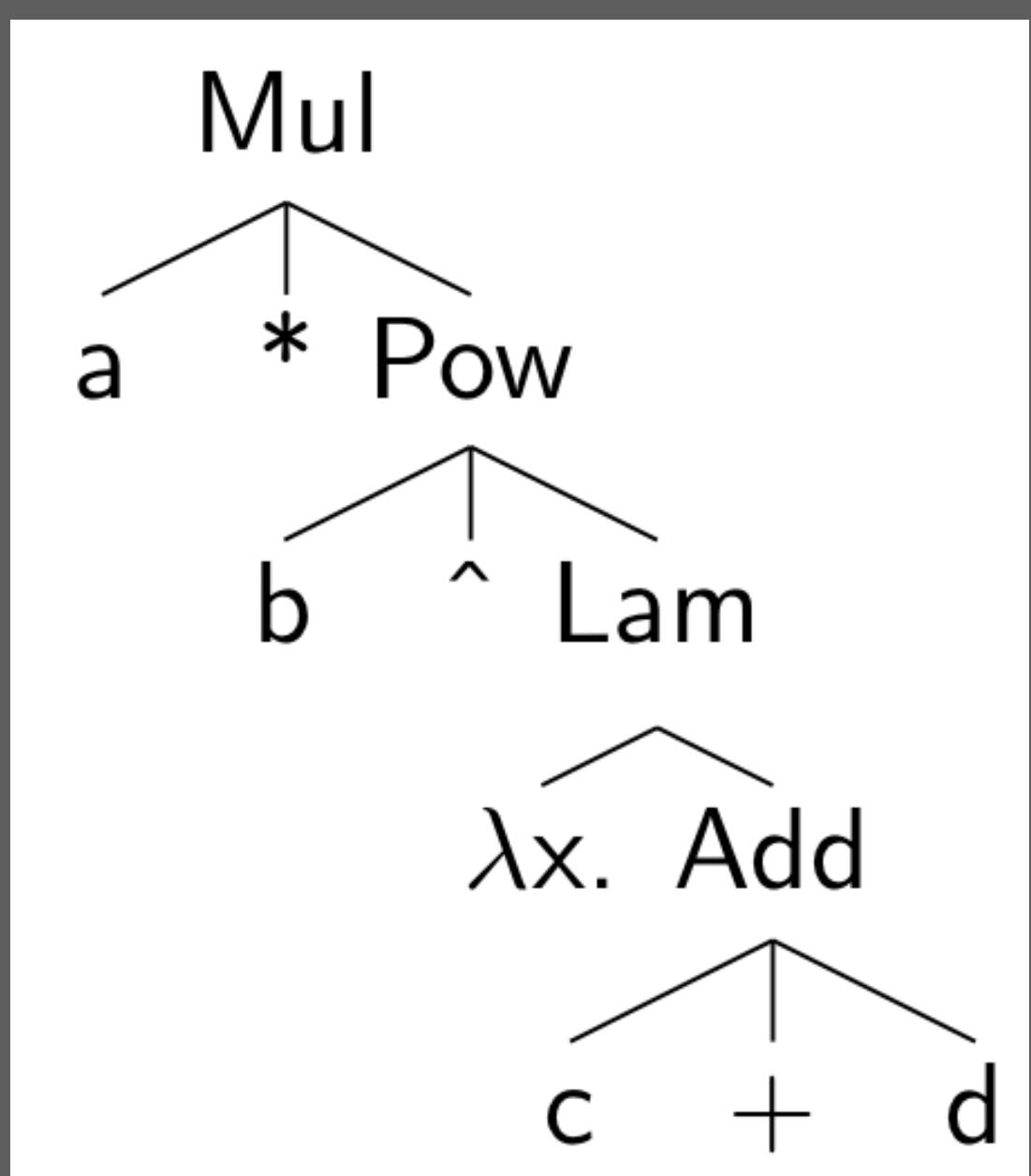
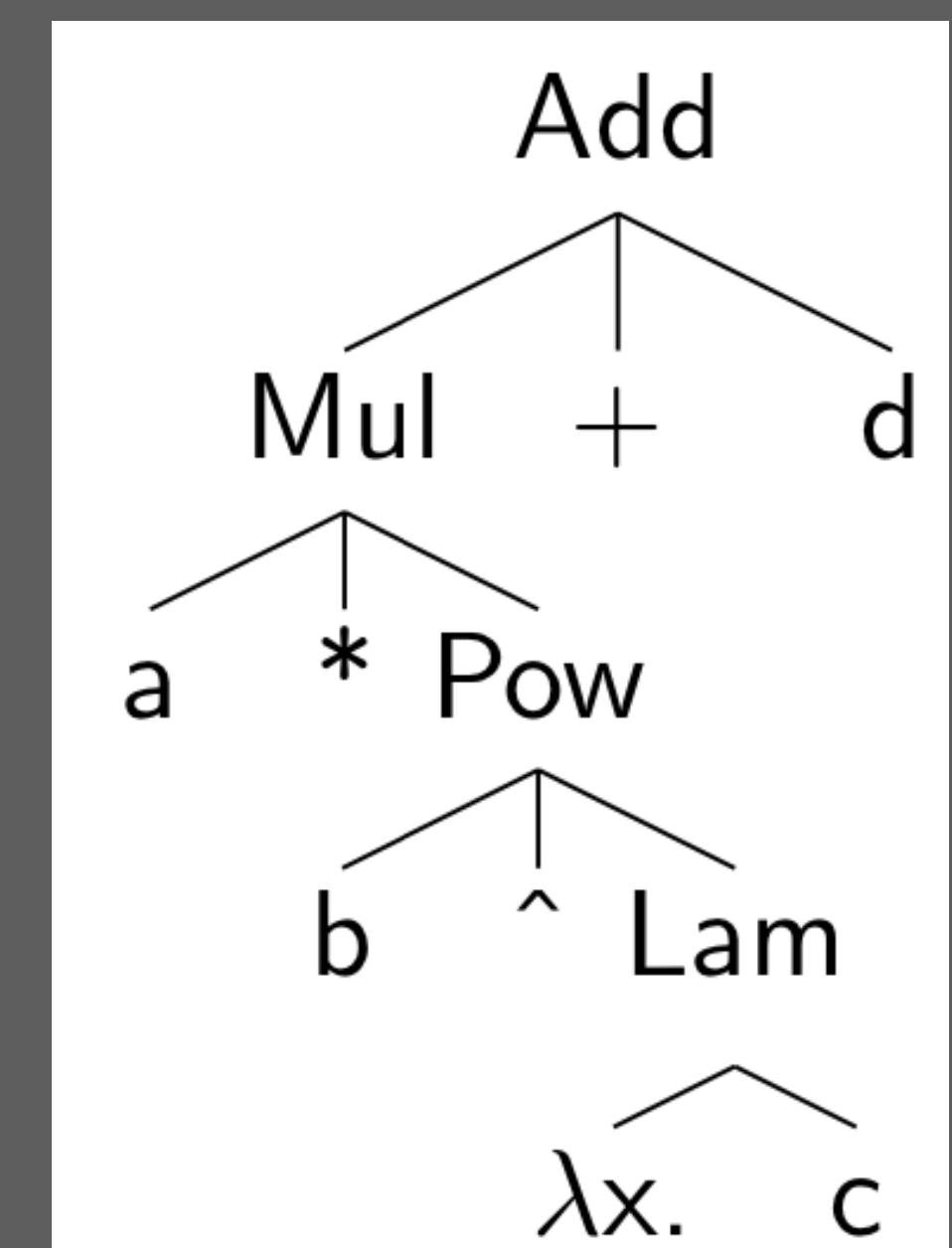
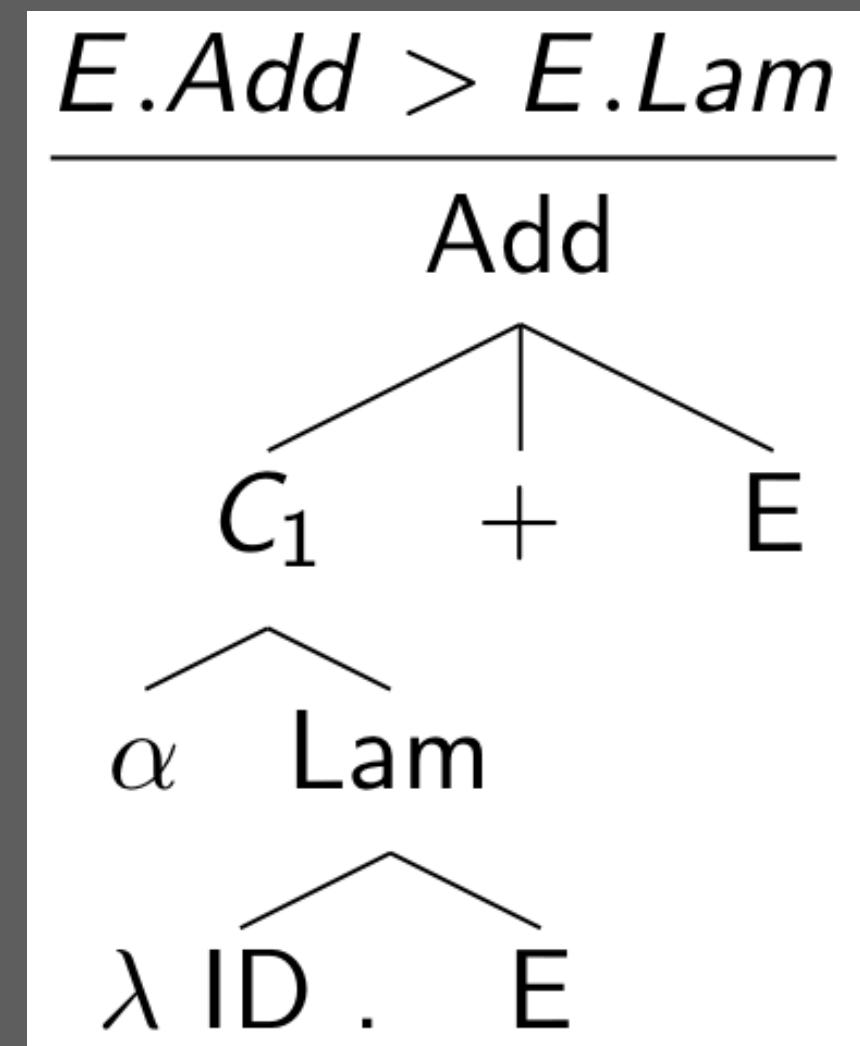
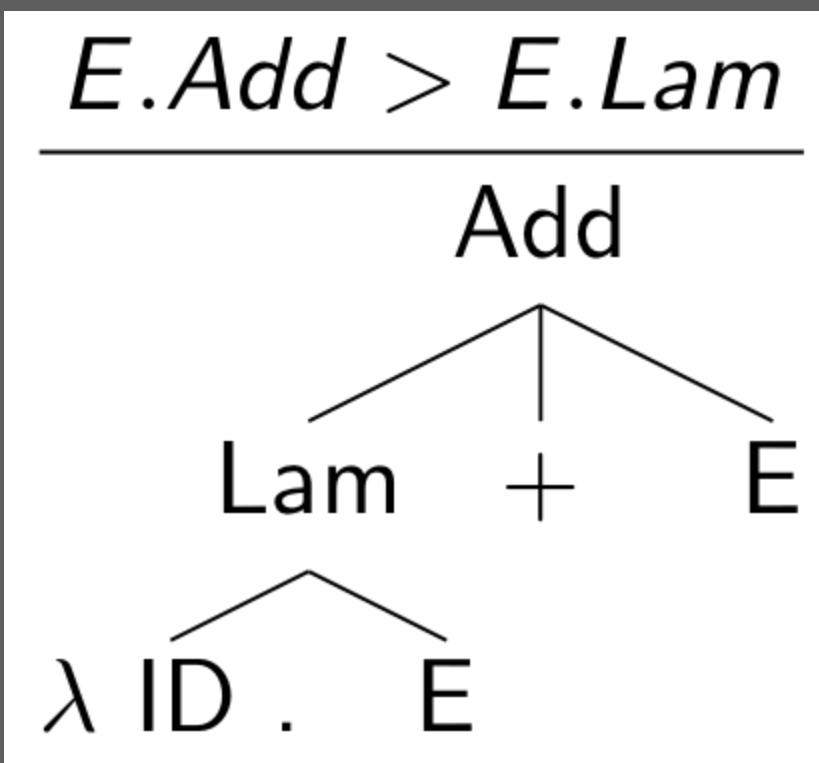
$a * b ^ \lambda x. c + d$



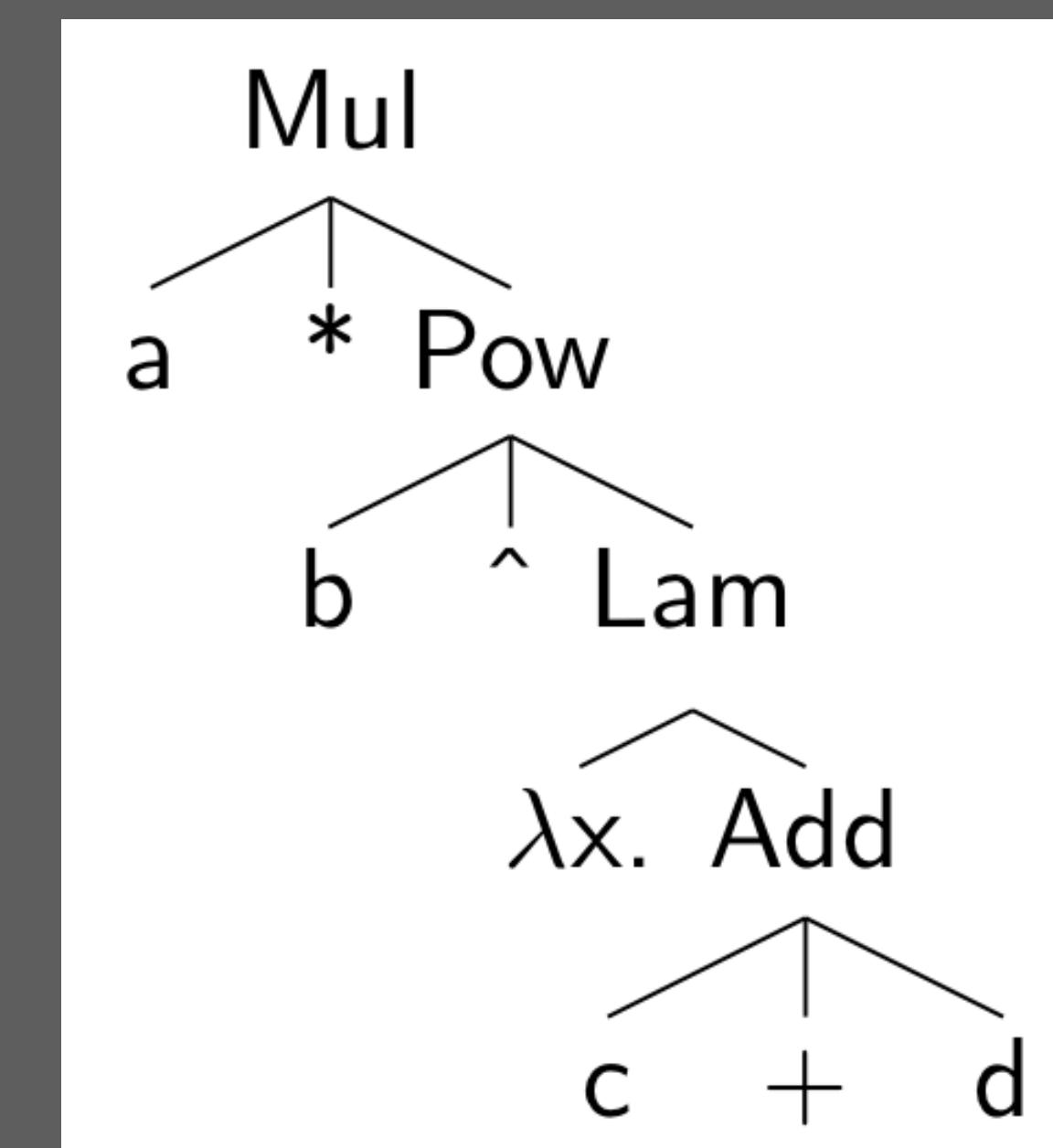
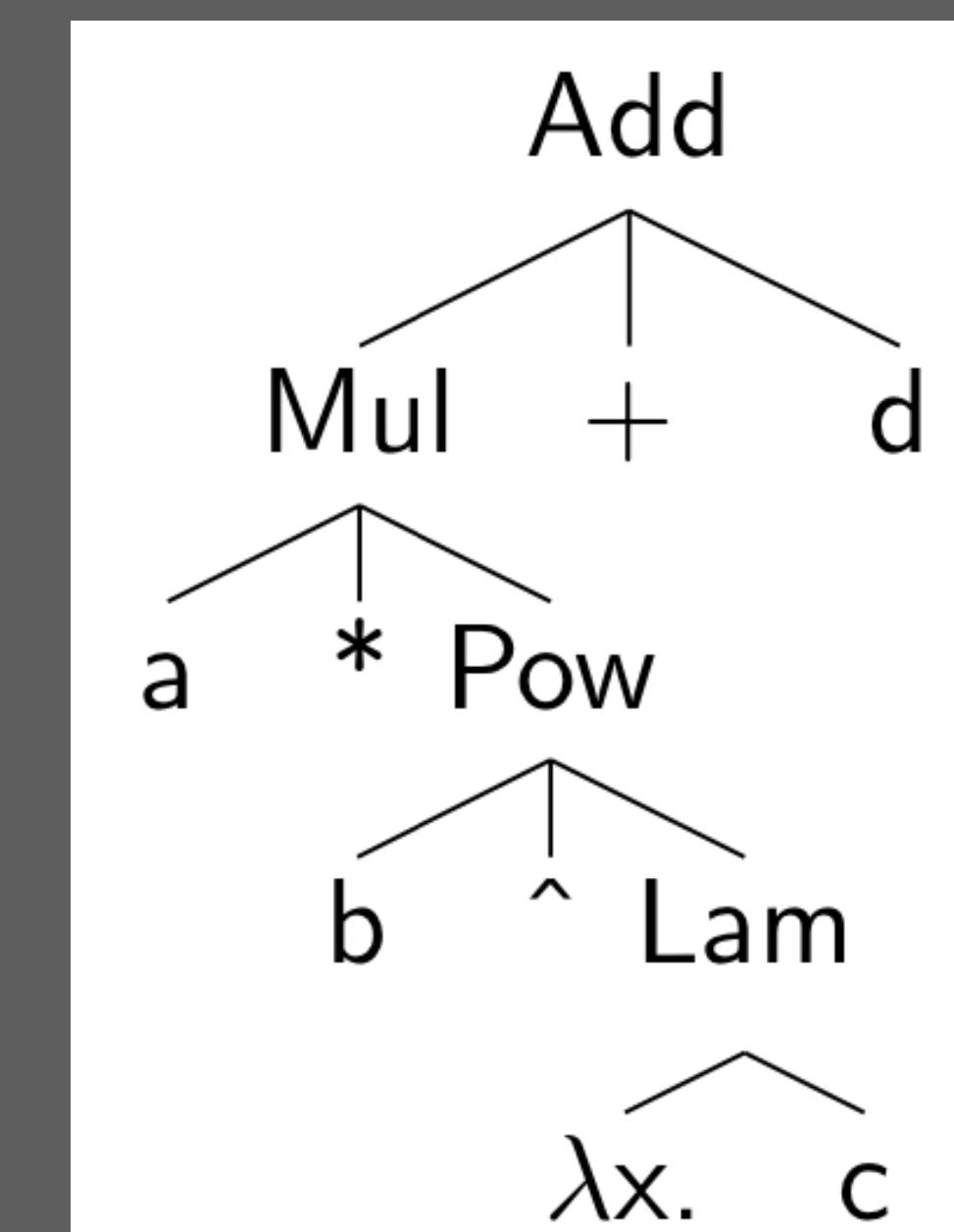
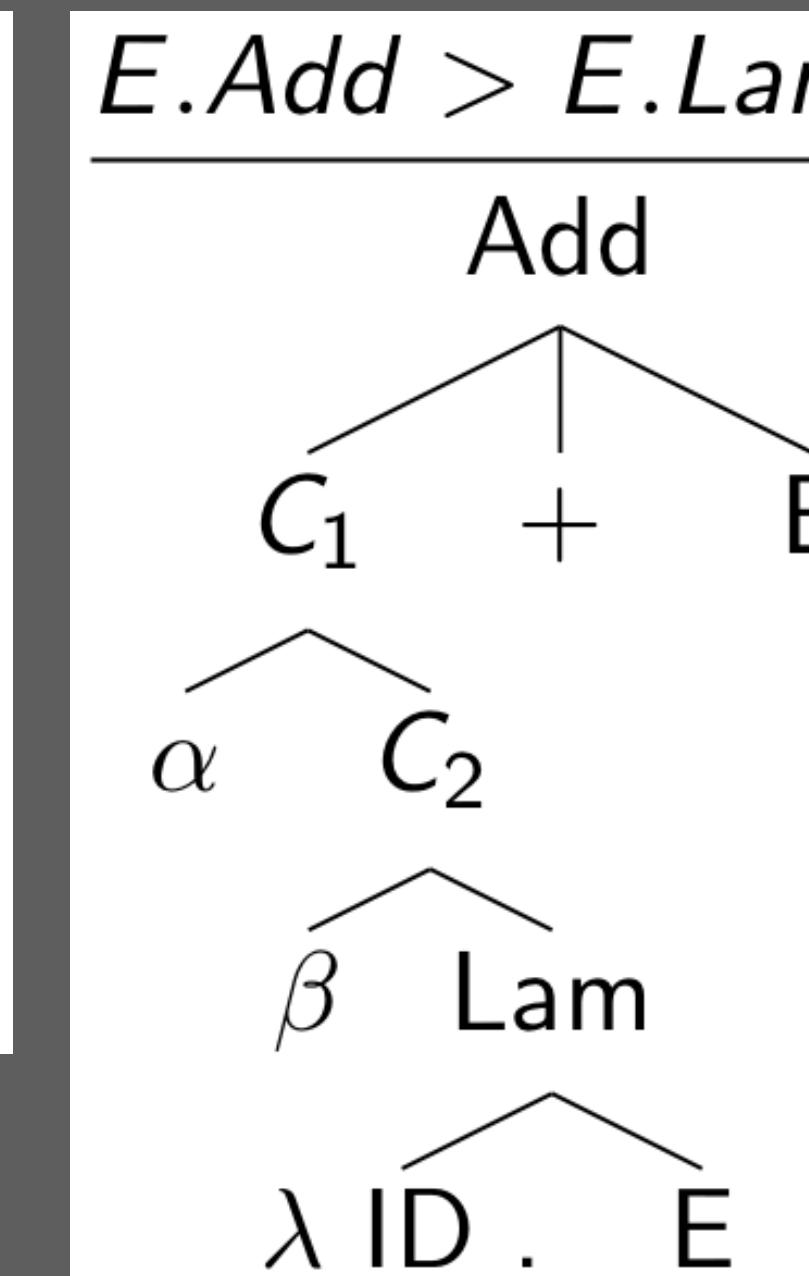
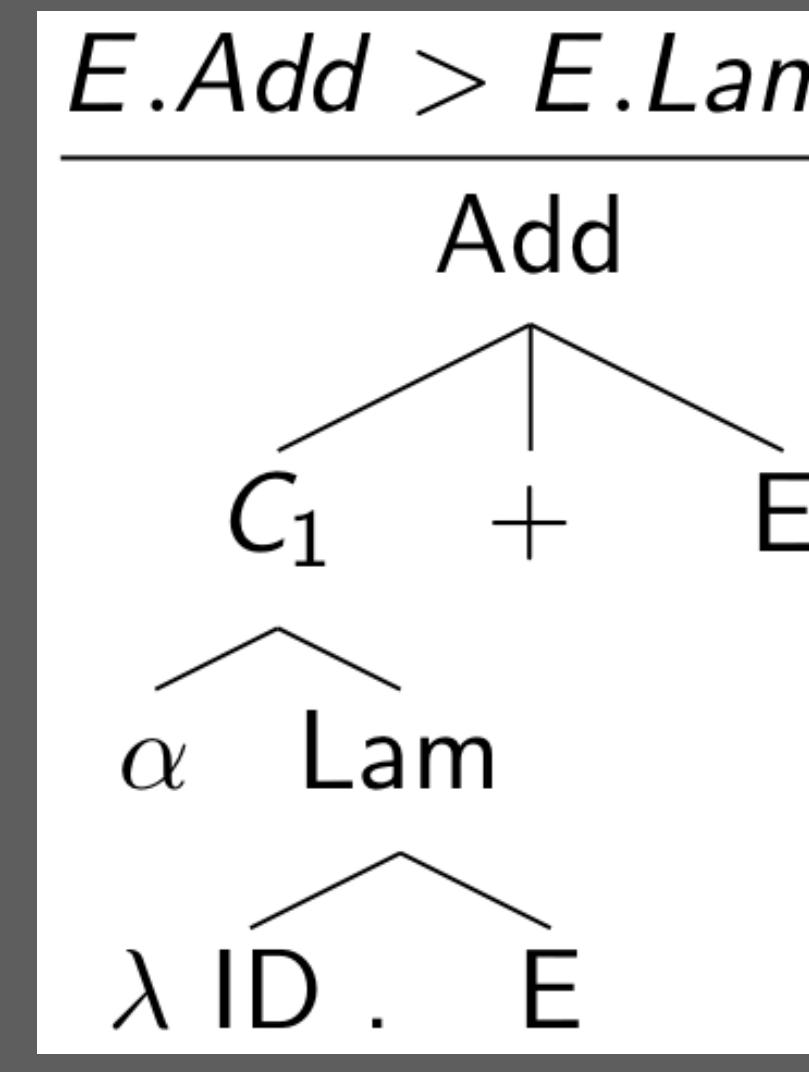
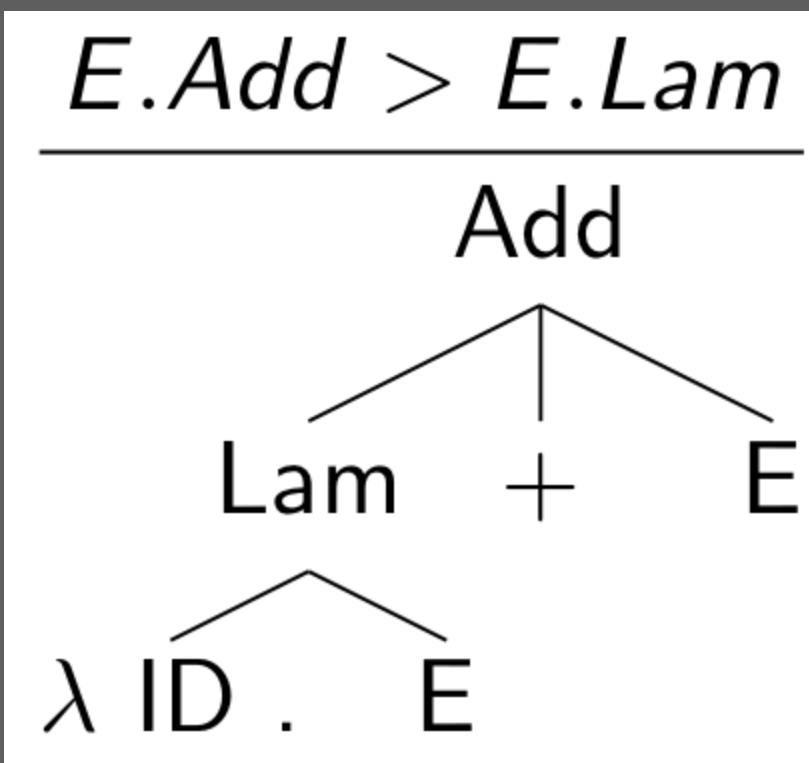
Deep Priority Conflicts: Match Subpattern in Right-Most Subtree



Deep Priority Conflicts: Match Subpattern in Right-Most Subtree



Deep Priority Conflicts: Match Subpattern in Right-Most Subtree



Deep Priority Conflicts: Match Subpattern in Right-Most Subtree

$$\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}$$

Tree diagram:

```
graph TD; Add --> Lam; Add --> Plus; Add --> E; Lam --> Lambda; Lam --> ID; Lambda --> Dot; Lambda --> E;
```

$$\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}$$

Tree diagram:

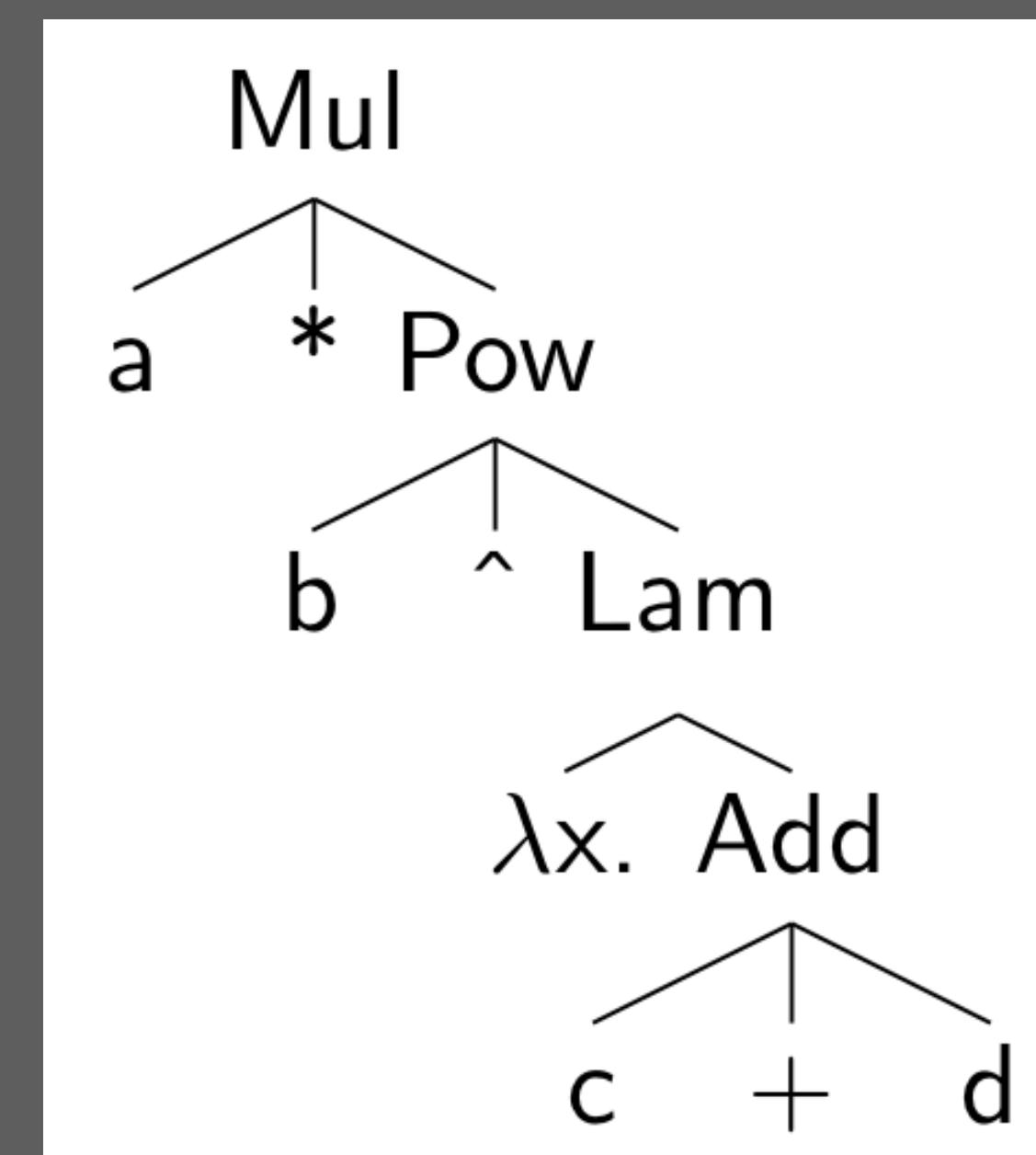
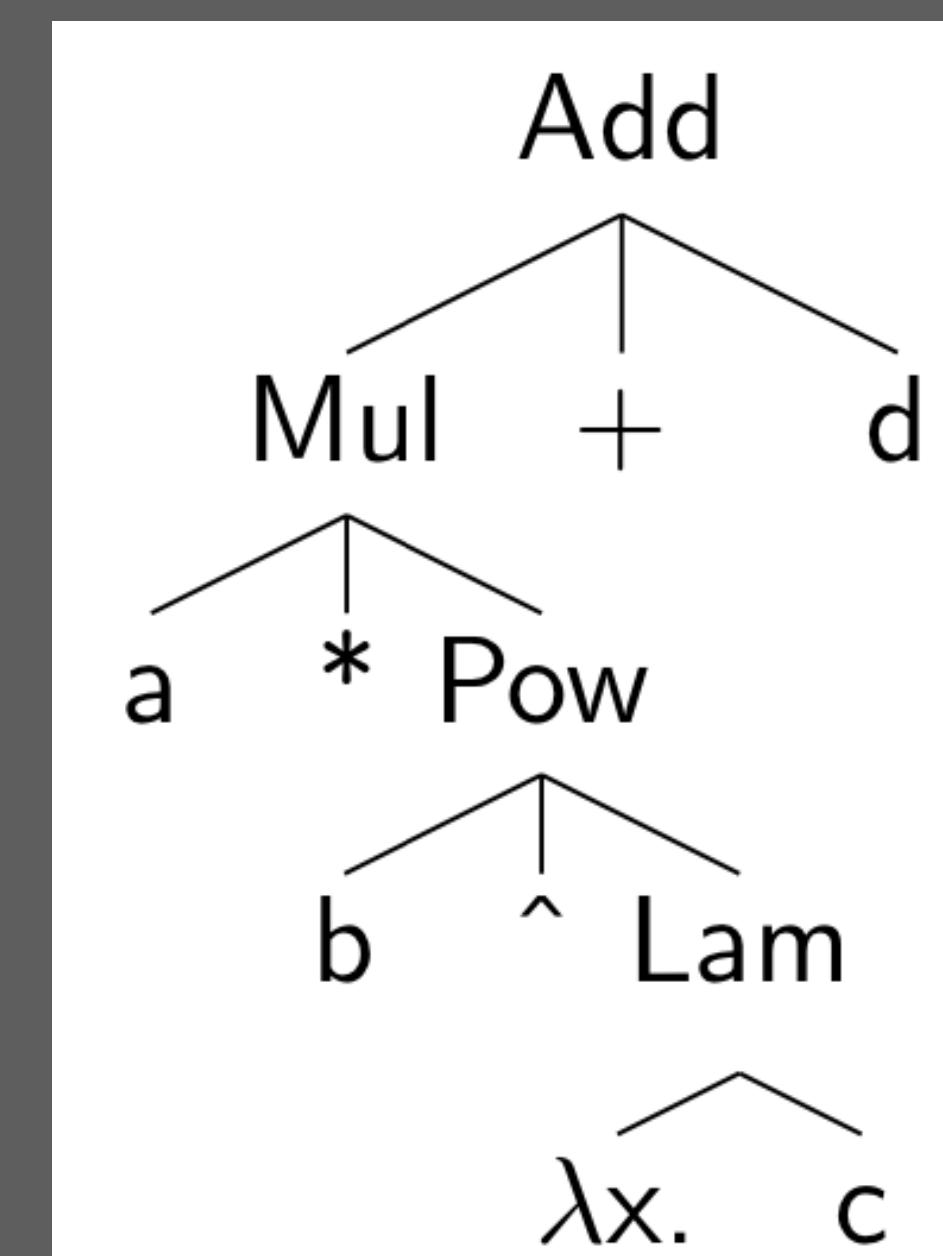
```
graph TD; Add --> C1; Add --> Plus; Add --> E; C1 --> Alpha; C1 --> Lam; Alpha --> Lambda; Alpha --> ID; Lambda --> Dot; Lambda --> E;
```

$$\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}$$

Tree diagram:

```
graph TD; Add --> C1; Add --> Plus; Add --> E; C1 --> Alpha; C1 --> C2; Alpha --> Beta; Alpha --> Lam; Beta --> Lambda; Beta --> ID; Lambda --> Dot; Lambda --> E;
```

...



Infinite set of conflict patterns

Deep Priority Conflicts: Match Subpattern in Right-Most Subtree

$$\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}$$

Tree diagram:

```
graph TD; Add --> Lam; Add --> Plus; Add --> E; Lam --> Lambda; Lam --> ID; Lambda --> Dot; Lambda --> E;
```

$$\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}$$

Tree diagram:

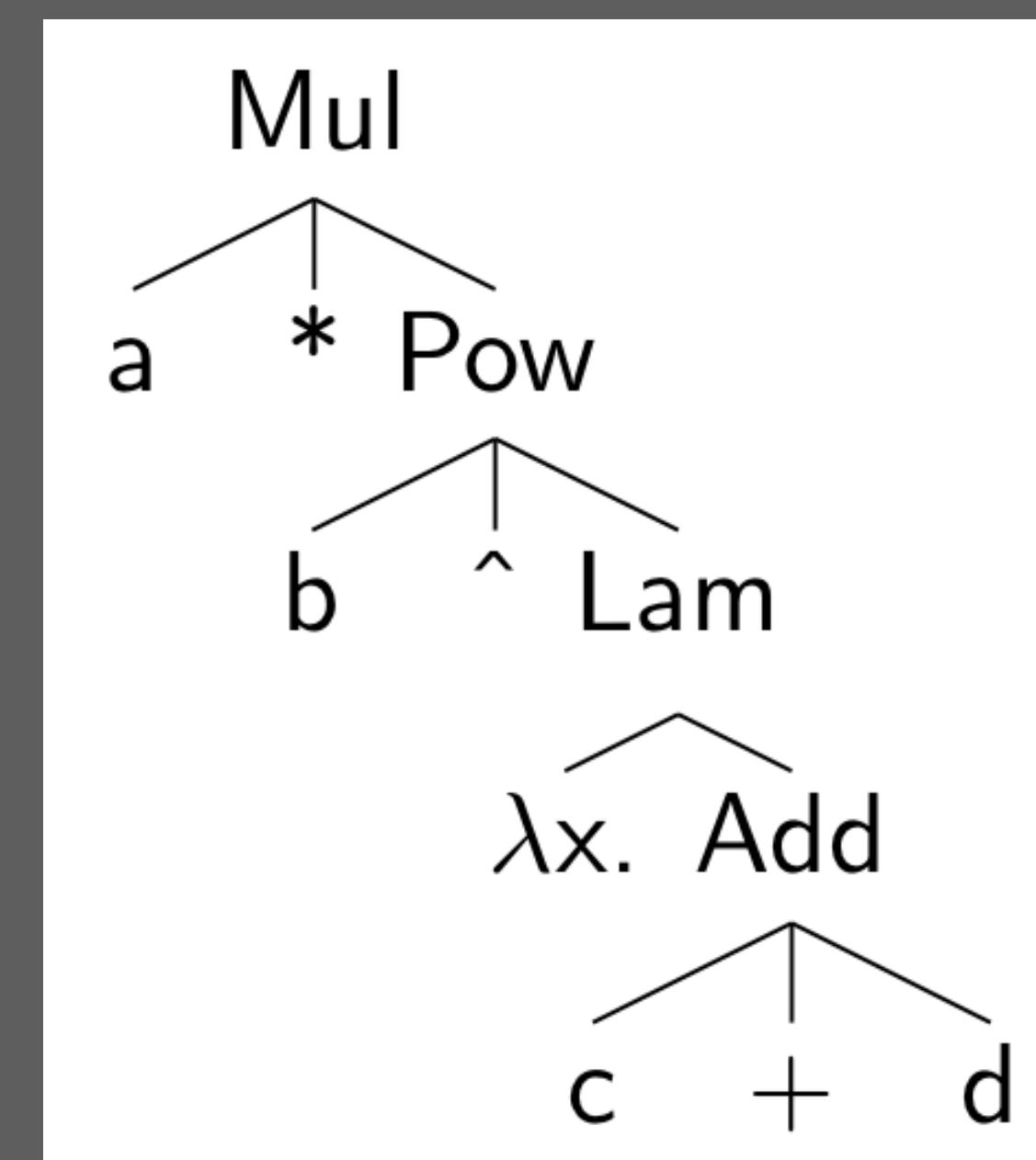
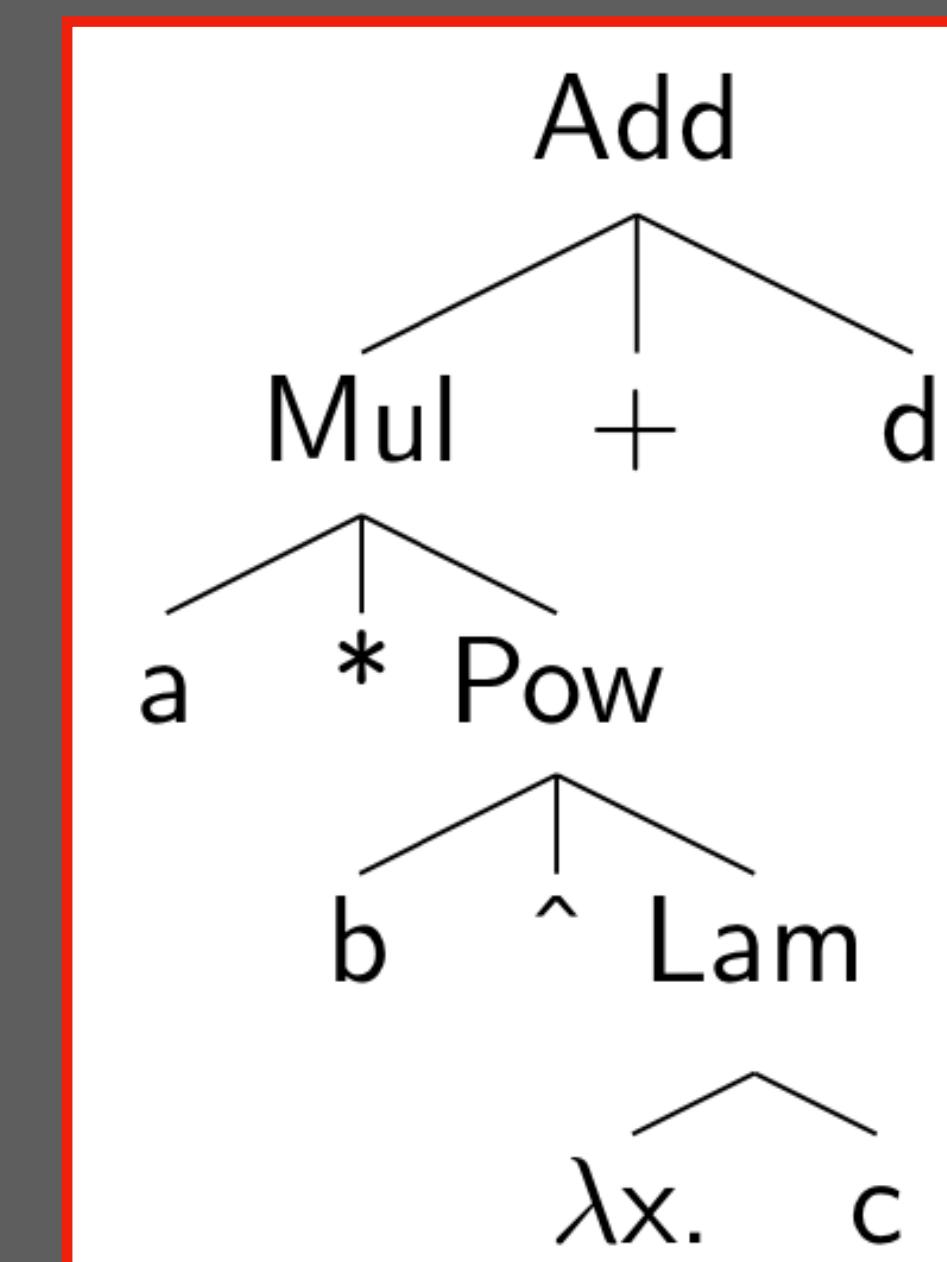
```
graph TD; Add --> C1; Add --> Plus; Add --> E; C1 --> Alpha; C1 --> Lam; Alpha --> Lambda; Alpha --> ID; Lambda --> Dot; Lambda --> E;
```

$$\frac{E.\text{Add} > E.\text{Lam}}{\text{Add}}$$

Tree diagram:

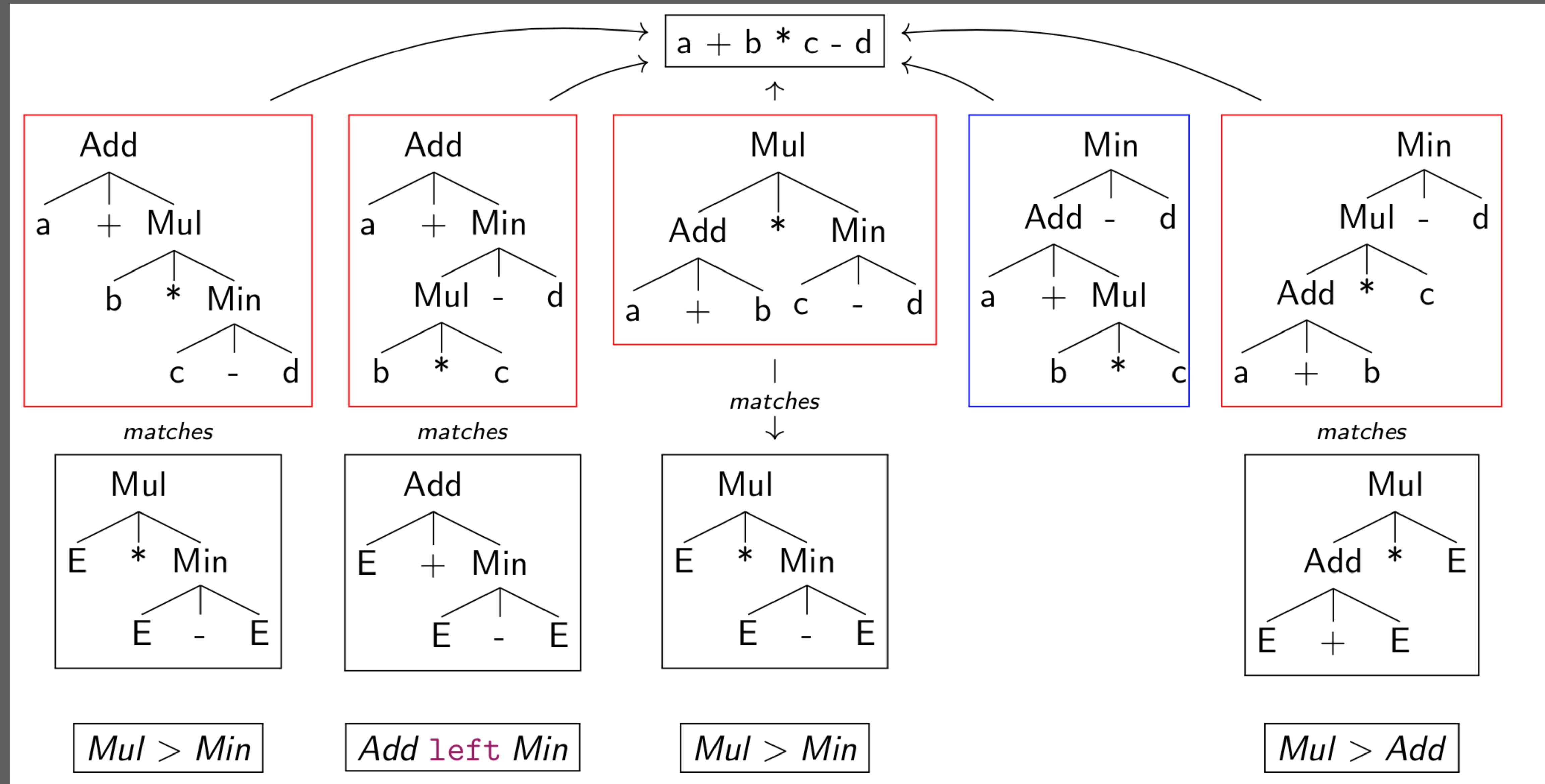
```
graph TD; Add --> C1; Add --> Plus; Add --> E; C1 --> Alpha; C1 --> C2; Alpha --> Beta; Alpha --> Lam; Beta --> Lambda; Beta --> ID; Lambda --> Dot; Lambda --> E;
```

...

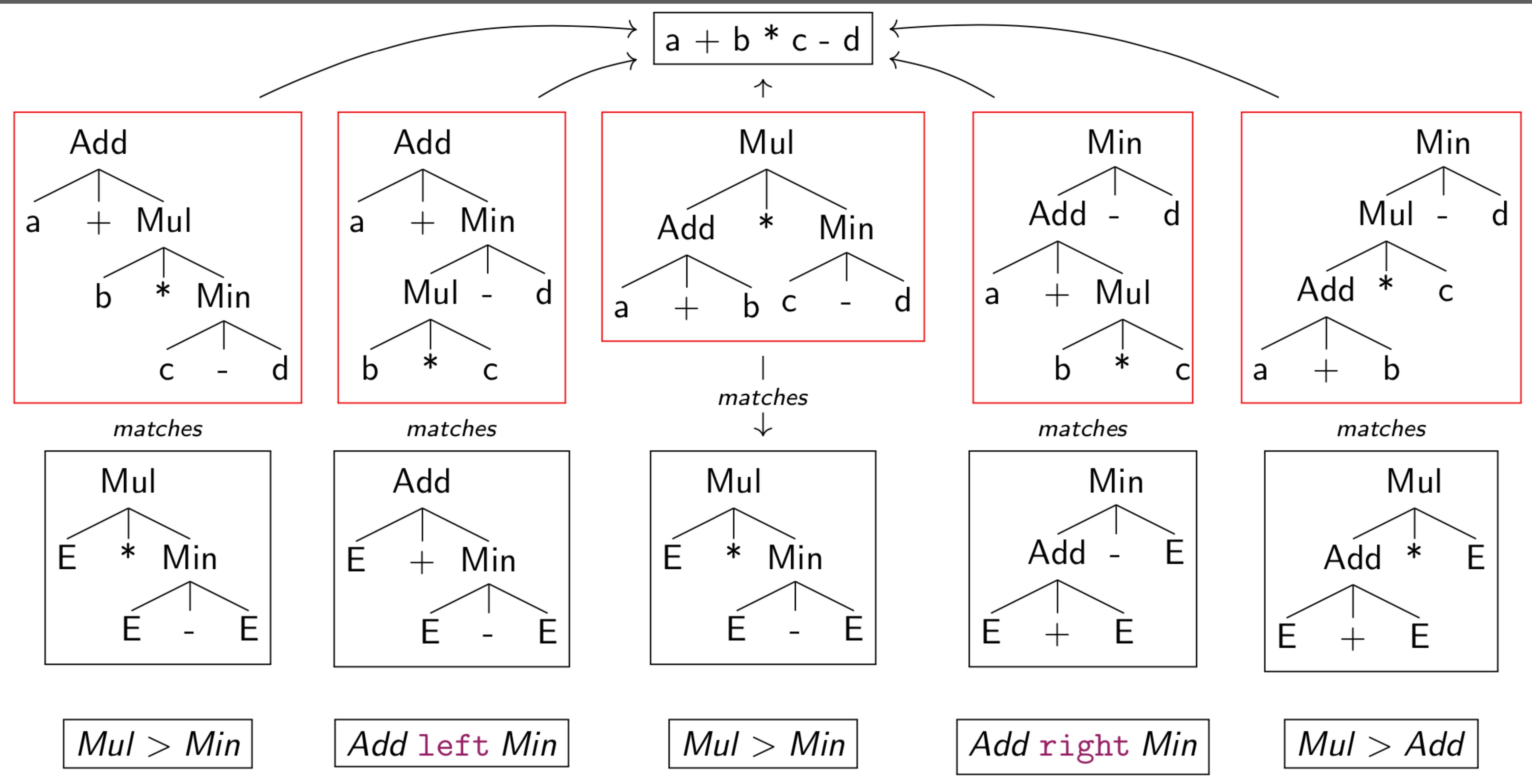


Infinite set of conflict patterns

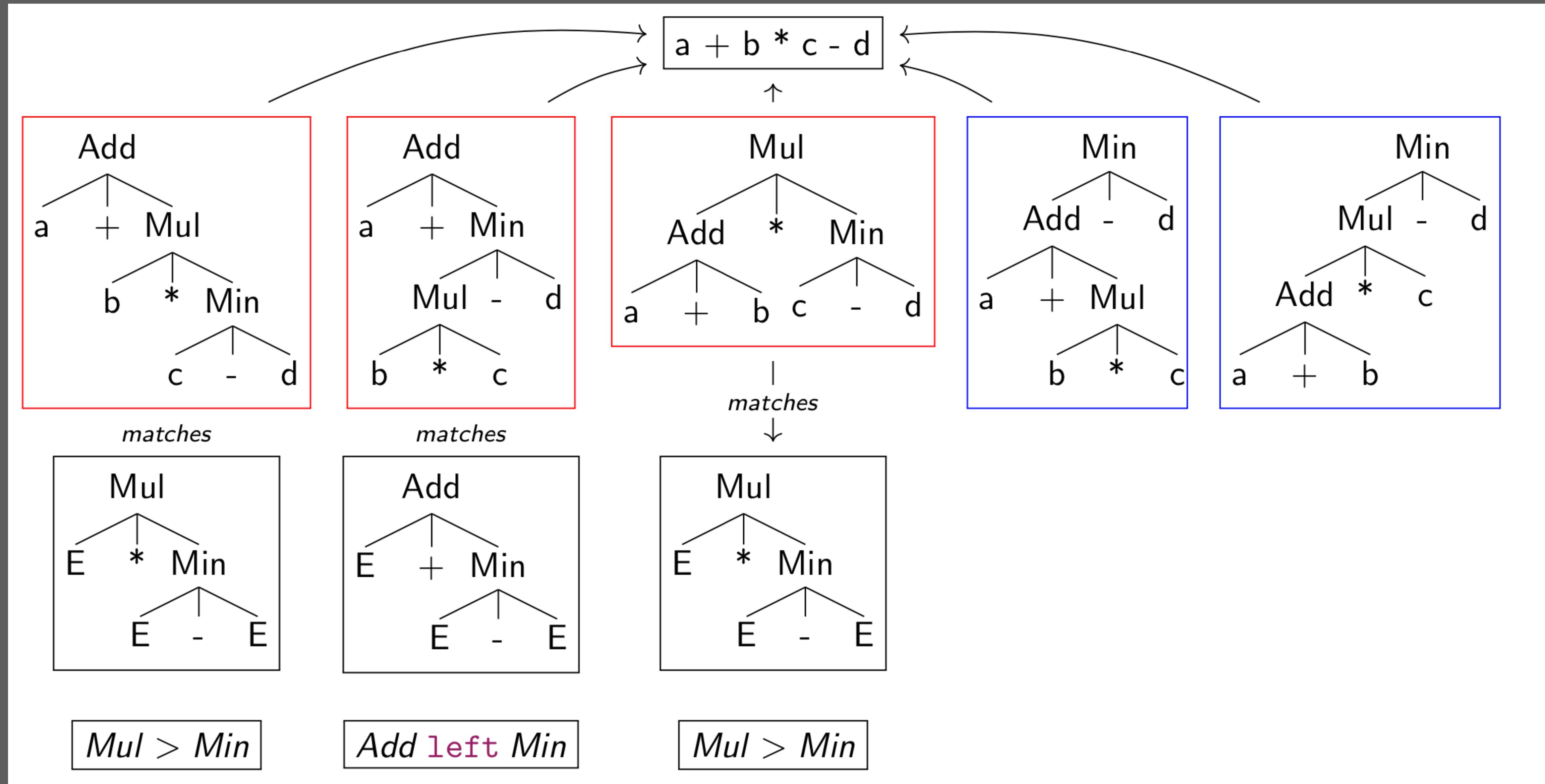
Safe and Complete Disambiguation Rules



Unsafe: Too Many Disambiguation Rules



Incomplete: Too Few Disambiguation Rules



Semantics of Associativity and Priority

What is the semantics of associativity and priority rules?

- subtree exclusion: tree patterns that are forbidden

Is a set of disambiguation rules safe?

- At most one rule for each pair of productions
- + some well-formedness criteria

Is a set of disambiguation rules complete?

- At least one rule for each pair of productions
- + some well-formedness criteria

How to implement?

Grammar Transformations

Grammar Transformations

Why?

- Disambiguation
- For use by a particular parsing algorithm

Transformations

- Eliminating ambiguities
- Eliminating left recursion
- Left factoring

Properties

- Does transformation preserve the language (set of strings, trees)?
- Does transformation preserve the structure of trees?

Ambiguous Expression Grammar

grammar

productions

$E.A = E "+" E$
 $E.T = E "*" E$
 $E.M = "-" E$
 $E.B = "(" E ")"$
 $E.V = ID$

derivation

$E \Rightarrow^* ID "*" ID "+" ID$

term derivation

E
 $\Rightarrow A(E, E)$
 $\Rightarrow A(T(E, E), E)$
 $\Rightarrow A(T(E, E), E)$
 $\Rightarrow A(T(V(ID), E), E)$
 $\Rightarrow A(T(V(ID), V(ID)), E)$
 $\Rightarrow A(T(V(ID), V(ID)), V(ID))$

term derivation

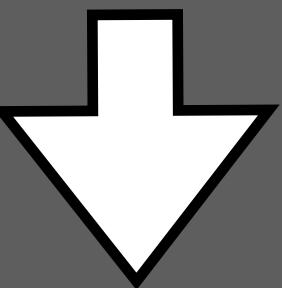
E
 $\Rightarrow T(E, E)$
 $\Rightarrow T(E, E)$
 $\Rightarrow T(V(ID), E)$
 $\Rightarrow T(V(ID), A(E, E))$
 $\Rightarrow T(V(ID), A(V(ID), E))$
 $\Rightarrow T(V(ID), A(V(ID), V(ID)))$

Associativity and Priority Filter Ambiguities

grammar

productions

```
E.A = E "+" E  
E.T = E "*" E  
E.M = "-" E  
E.B = "(" E ")"  
E.V = ID
```



derivation

$E \Rightarrow^* ID \ast ID + ID$

term derivation

```
E  
⇒ A(E, E)  
⇒ A(T(E, E), E)  
⇒ A(T(E, E), E)  
⇒ A(T(V(ID), E), E)  
⇒ A(T(V(ID), V(ID)), E)  
⇒ A(T(V(ID), V(ID)), V(ID))
```

grammar

productions

```
E.A = E "+" E {left}  
E.T = E "*" E {left}  
E.M = "-" E  
E.B = "(" E ")"  
E.V = ID
```

priorities

$E.M > E.T > E.A$

term derivation

```
E  
⇒ T(E, E)  
⇒ T(E, E)  
⇒ T(V(ID), E)  
⇒ T(V(ID), A(E, E))  
⇒ T(V(ID), A(V(ID), E))  
⇒ T(V(ID), A(V(ID), V(ID)))
```

Define Associativity and Priority by Transformation

grammar

productions

$E.A = E "+" E \{left\}$

$E.T = E "*" E \{left\}$

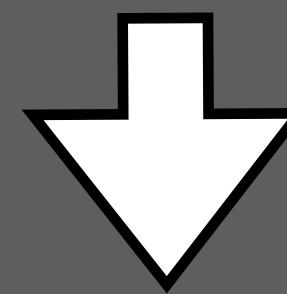
$E.M = "-" E$

$E.B = "(" E ")"$

$E.V = ID$

priorities

$E.M > E.T > E.A$



derivation

$E \Rightarrow^* ID "*" ID "+" ID$

term derivation

E

$\Rightarrow A(E, E)$

$\Rightarrow A(T(E, E), E)$

$\Rightarrow A(T(E, E), E)$

$\Rightarrow A(T(V(ID), E), E)$

$\Rightarrow A(T(V(ID), V(ID)), E)$

$\Rightarrow A(T(V(ID), V(ID)), V(ID))$

grammar

productions

$E.A = E "+" T$

$E = T$

$T.T = T "*" F$

$T = F$

$F.V = ID$

$F.B = "(" E ")"$

term derivation

E

$\Rightarrow T(E, E)$

$\Rightarrow T(E, E)$

$\Rightarrow T(V(ID), E)$

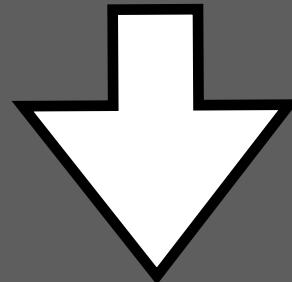
$\Rightarrow T(V(ID), A(E, E))$

$\Rightarrow T(V(ID), A(V(ID), E))$

$\Rightarrow T(V(ID), A(V(ID), V(ID)))$

Define Associativity and Priority by Transformation

```
grammar  
productions  
E.A = E "+" E {left}  
E.T = E "*" E {left}  
E.M = "-" E  
E.B = "(" E ")"  
E.V = ID  
priorities  
E.M > E.T > E.A
```



```
grammar  
productions  
E.A = E "+" T  
E = T  
T.T = T "*" F  
T = F  
F.V = ID  
F.M = "-" F  
F.B = "(" E ")"
```

Define new non-terminal for each priority level:

E, T, F

Add 'injection' productions to include priority level n+1 in n:

E = T
T = F

Change head of production to reflect priority level

T = T "*" F

Transform productions

Left: E = E "+" T
Right: E = T "+" E

Dangling Else Grammar

grammar

sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other

derivation

$S \Rightarrow^* \text{if } E_1 \text{ then } S_1 \text{ else if } E_2 \text{ then } S_2 \text{ else } S_3$

term derivation

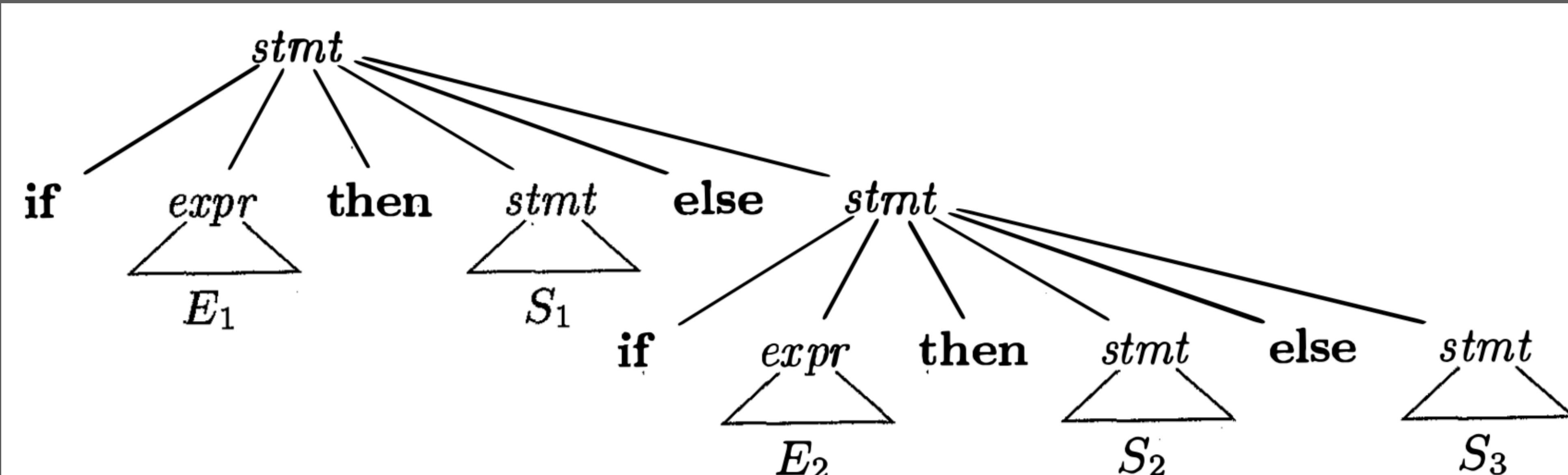
$S \Rightarrow^* \text{IfE}(E_1, S_1, \text{IfE}(E_2, S_2, S_3))$

term derivation

S

$\Rightarrow \text{IfE}(E_1, S_1, S)$

$\Rightarrow \text{IfE}(E_1, S_1, \text{IfE}(E_2, S_2, S_3))$



Dangling Else Grammar is Ambiguous

grammar

sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other

derivation

$S \Rightarrow^* \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

Dangling Else Grammar is Ambiguous

grammar

sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other

derivation

$S \Rightarrow^* \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

term derivation

S

$\Rightarrow \text{If}(E_1, S)$

$\Rightarrow \text{If}(E_1, \text{IfE}(E_2, S_1, S_2))$

derivation

S

$\Rightarrow \text{if } E_1 \text{ then } S$

$\Rightarrow \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

Dangling Else Grammar is Ambiguous

grammar

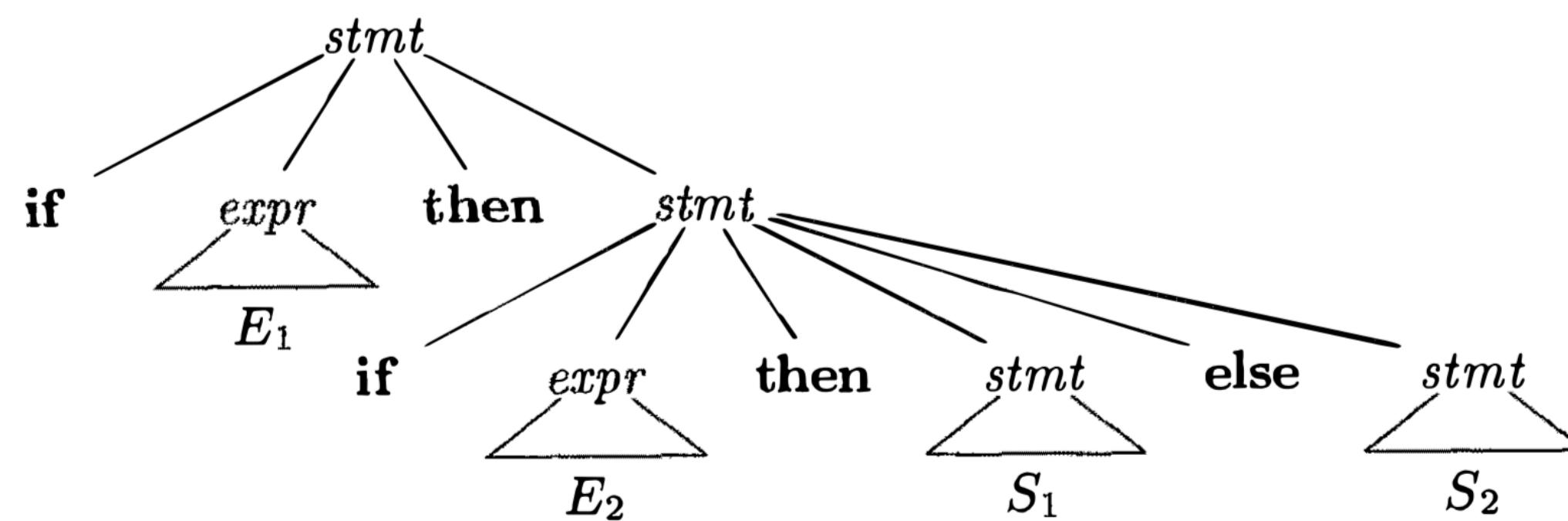
sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other



derivation

$S \Rightarrow^* \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

term derivation

$S \Rightarrow \text{If}(E_1, S)$

$\Rightarrow \text{If}(E_1, \text{IfE}(E_2, S_1, S_2))$

derivation

$S \Rightarrow \text{if } E_1 \text{ then } S$

$\Rightarrow \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

Dangling Else Grammar is Ambiguous

grammar

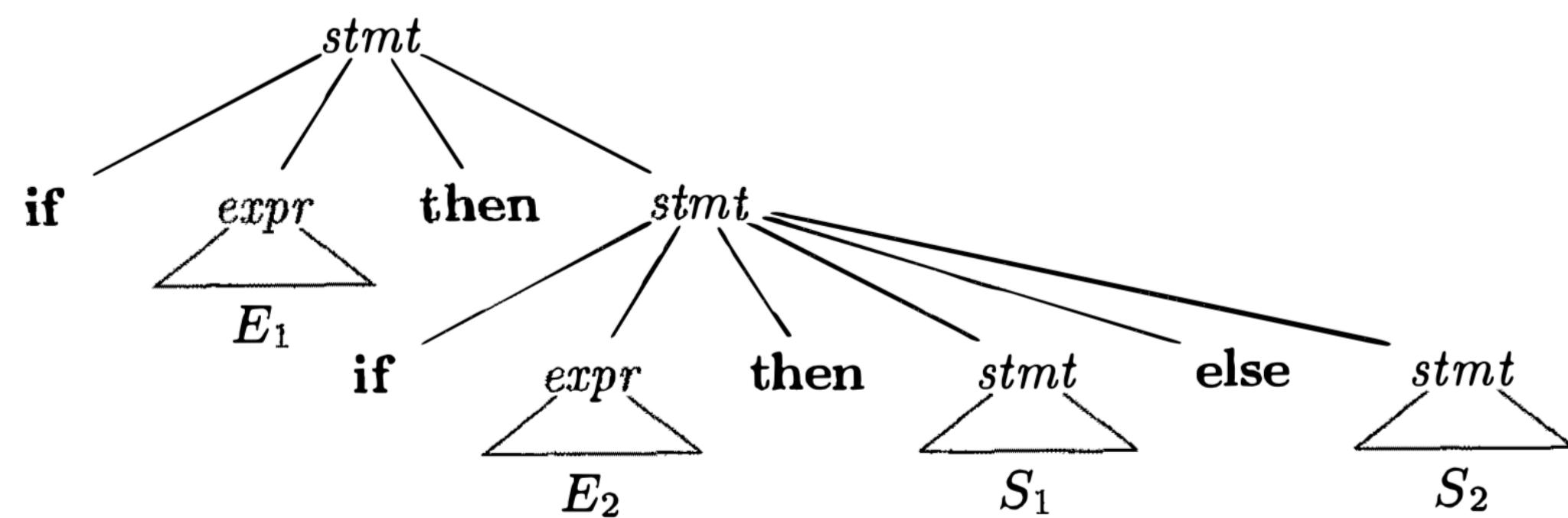
sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other



derivation

$S \Rightarrow^* \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

term derivation

$S \Rightarrow \text{If}(E_1, S)$

$\Rightarrow \text{If}(E_1, \text{IfE}(E_2, S_1, S_2))$

derivation

$S \Rightarrow \text{if } E_1 \text{ then } S$

$\Rightarrow \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

term derivation

$S \Rightarrow \text{IfE}(E_1, S, S_2)$

$\Rightarrow \text{IfE}(E_1, \text{If}(E_2, S_1), S_2)$

derivation

$S \Rightarrow \text{if } E_1 \text{ then } S \text{ else } S_2$

$\Rightarrow \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

Dangling Else Grammar is Ambiguous

grammar

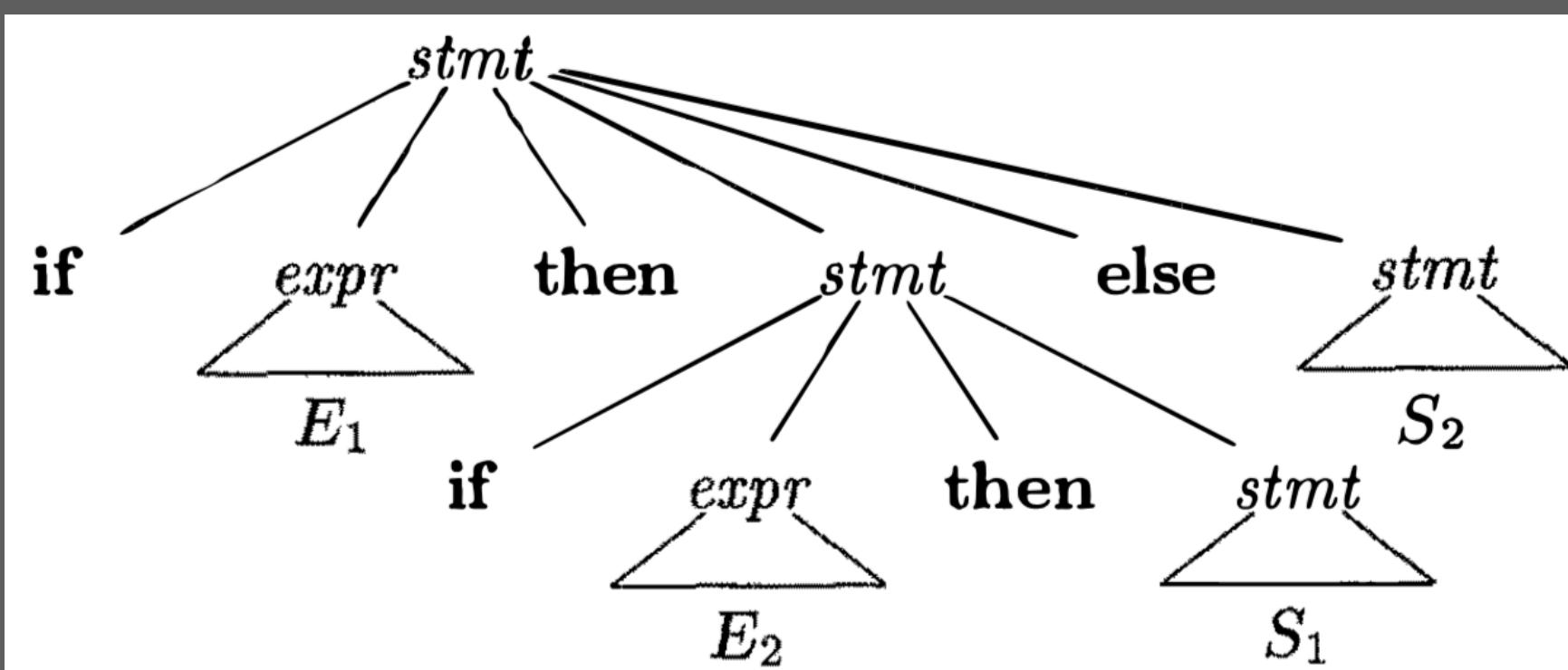
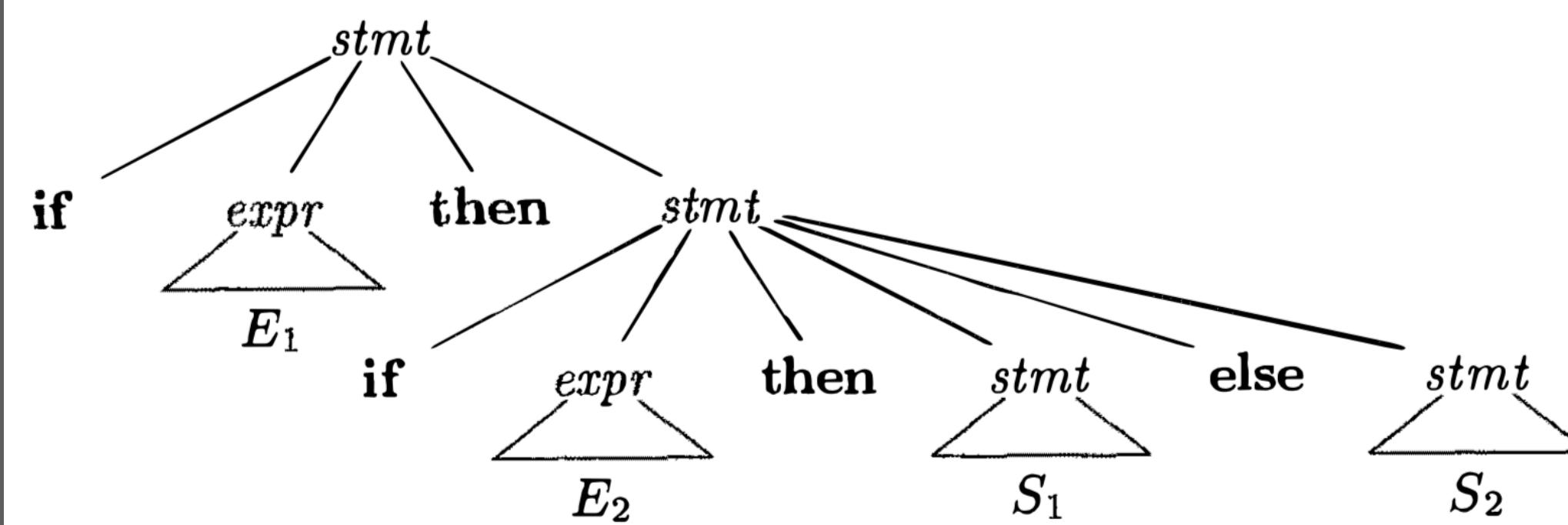
sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other



derivation

$S \Rightarrow^* \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

term derivation

$S \Rightarrow \text{If}(E_1, S)$
 $\Rightarrow \text{If}(E_1, \text{IfE}(E_2, S_1, S_2))$

derivation

$S \Rightarrow \text{if } E_1 \text{ then } S$
 $\Rightarrow \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

term derivation

$S \Rightarrow \text{IfE}(E_1, S, S_2)$
 $\Rightarrow \text{IfE}(E_1, \text{If}(E_2, S_1), S_2)$

derivation

$S \Rightarrow \text{if } E_1 \text{ then } S \text{ else } S_2$
 $\Rightarrow \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$

Eliminating Dangling Else Ambiguity

grammar

sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other

Eliminating Dangling Else Ambiguity

grammar

sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other

grammar

productions

S.If = if E then S

S.IfE = if E then SE else S

S = other

SE.IfE = if E then SE else SE

SE = other

Eliminating Dangling Else Ambiguity

grammar

sorts S E

productions

S.If = if E then S

S.IfE = if E then S else S

S = other

grammar

productions

S.If = if E then S

S.IfE = if E then SE else S

S = other

SE.IfE = if E then SE else SE

SE = other

Generalization of this transformation: contextual grammars

Eliminating Left Recursion

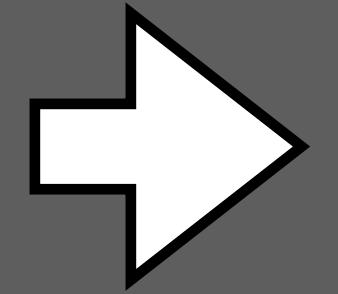
grammar
productions

```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```

Eliminating Left Recursion

grammar
productions

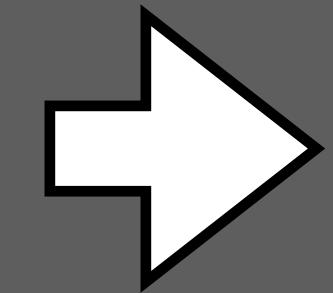
```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```



Eliminating Left Recursion

grammar
productions

```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```



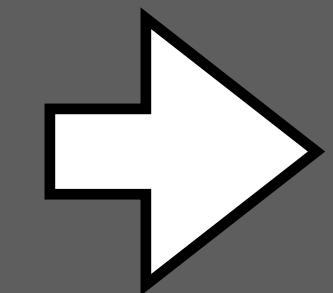
grammar
productions

```
E = T E'
E' = "+" T E'
E' =
T = F T'
T' = "*" F T'
T' =
F = "(" E ")"
F = ID
```

Eliminating Left Recursion

grammar
productions

```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```



grammar
productions

```
E = T E'
E' = "+" T E'
E' =
T = F T'
T' = "*" F T'
T' =
F = "(" E ")"
F = ID
```

grammar
productions

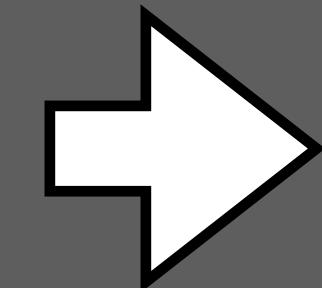
```
A = A a
A = b
```

// b followed by a list of as

Eliminating Left Recursion

grammar
productions

```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```

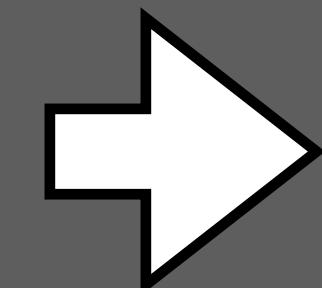


grammar
productions

```
E = T E'
E' = "+" T E'
E' =
T = F T'
T' = "*" F T'
T' =
F = "(" E ")"
F = ID
```

grammar
productions

```
A = A a
A = b
```

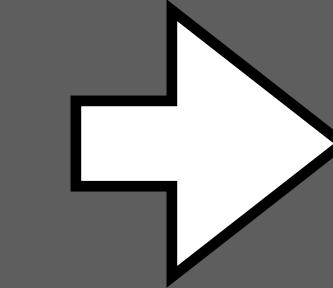


// b followed by a list of as

Eliminating Left Recursion

grammar
productions

$E = E \text{ "+" } T$
 $E = T$
 $T = T \text{ "*" } F$
 $T = F$
 $F = "(" \text{ } E \text{ } ")"$
 $F = \text{ID}$

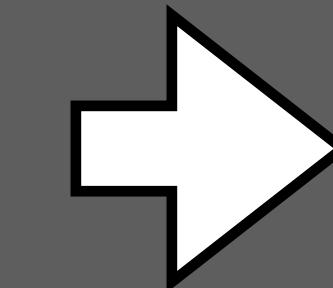


grammar
productions

$E = T \text{ } E'$
 $E' = \text{"+" } T \text{ } E'$
 $E' =$
 $T = F \text{ } T'$
 $T' = \text{"*"} \text{ } F \text{ } T'$
 $T' =$
 $F = "(" \text{ } E \text{ } ")"$
 $F = \text{ID}$

grammar
productions

$A = A \text{ } a$
 $A = b$



grammar
productions

$A = b \text{ } A'$
 $A' = a \text{ } A'$
 $A' = // \text{ empty}$

// b followed by a list of as

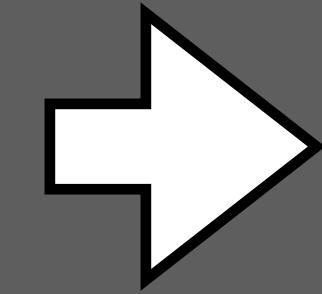
Eliminating Left Recursion using Regular Expressions

```
grammar  
productions  
E = E "+" T  
E = T  
T = T "*" F  
T = F  
F = "(" E ")"  
F = ID
```

Eliminating Left Recursion using Regular Expressions

grammar
productions

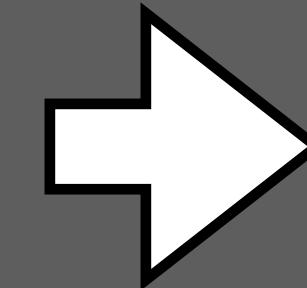
```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```



Eliminating Left Recursion using Regular Expressions

grammar
productions

```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```



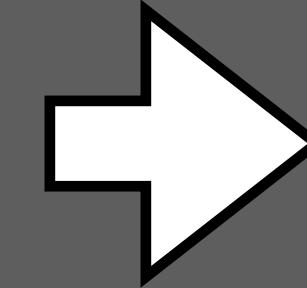
grammar
productions

```
E = T ("+" T)*
T = F ("*" F)*
F = "(" E ")"
F = ID
```

Eliminating Left Recursion using Regular Expressions

grammar
productions

```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```



grammar
productions

```
E = T ("+" T)*
T = F ("*" F)*
F = "(" E ")"
F = ID
```

grammar
productions

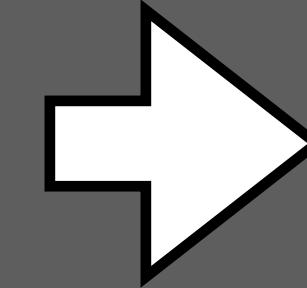
```
A = A a
A = b
```

// b followed by a list of as

Eliminating Left Recursion using Regular Expressions

grammar
productions

```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```

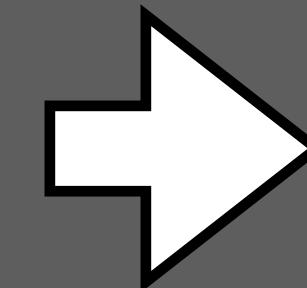


grammar
productions

```
E = T ("+" T)*
T = F ("*" F)*
F = "(" E ")"
F = ID
```

grammar
productions

```
A = A a
A = b
```

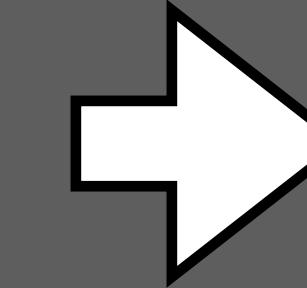


// b followed by a list of as

Eliminating Left Recursion using Regular Expressions

grammar
productions

```
E = E "+" T
E = T
T = T "*" F
T = F
F = "(" E ")"
F = ID
```

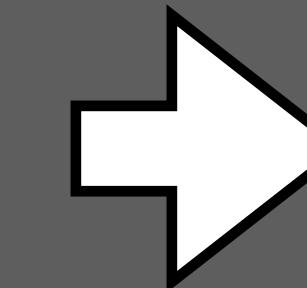


grammar
productions

```
E = T ("+" T)*
T = F ("*" F)*
F = "(" E ")"
F = ID
```

grammar
productions

```
A = A a
A = b
```



grammar
productions

```
A = b a*
```

// b followed by a list of as

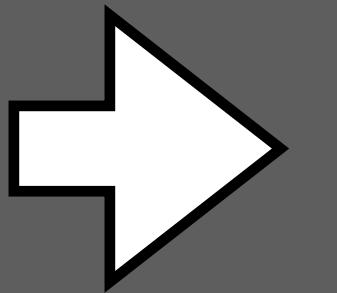
Left Factoring

```
grammar  
productions  
S.If  = if E then S  
S.IfE = if E then S else S  
S      = other
```

Left Factoring

grammar
productions

```
S.If  = if E then S
S.IfE = if E then S else S
S      = other
```

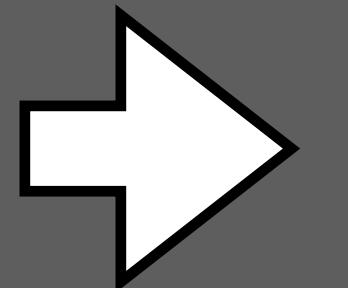


Left Factoring

grammar

productions

```
S.If    = if E then S  
S.IfE   = if E then S else S  
S       = other
```



grammar

sorts S E

productions

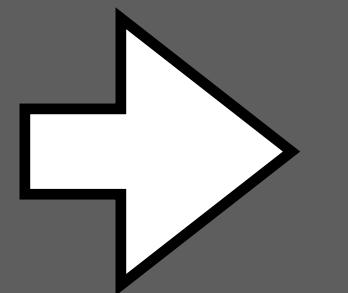
```
S.If      = if E then S S'  
S'.Else   = else S  
S'.NoElse = // empty  
S         = other
```

Left Factoring

grammar

productions

```
S.If    = if E then S  
S.IfE   = if E then S else S  
S       = other
```



grammar

sorts S E

productions

```
S.If      = if E then S S'  
S'.Else   = else S  
S'.NoElse = // empty  
S         = other
```

grammar

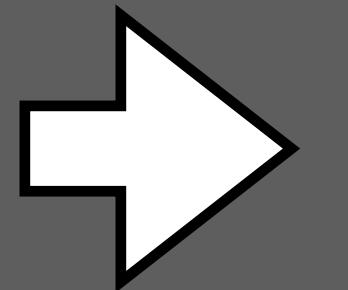
productions

```
A = a b1  
A = a b2  
A = c
```

Left Factoring

grammar
productions

```
S.If    = if E then S
S.IfE   = if E then S else S
S       = other
```

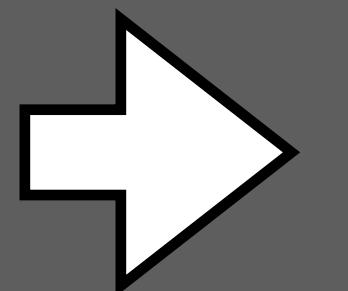


grammar
sorts S E
productions

```
S.If      = if E then S S'
S'.Else   = else S
S'.NoElse = // empty
S         = other
```

grammar
productions

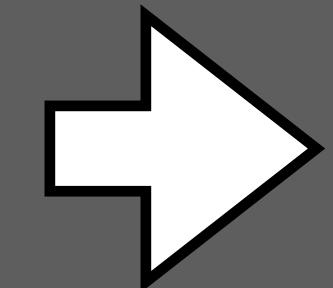
```
A = a b1
A = a b2
A = c
```



Left Factoring

grammar
productions

```
S.If    = if E then S
S.IfE   = if E then S else S
S       = other
```

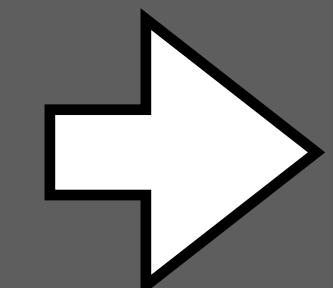


grammar
sorts S E
productions

```
S.If      = if E then S S'
S'.Else   = else S
S'.NoElse = // empty
S         = other
```

grammar
productions

```
A = a b1
A = a b2
A = c
```



grammar
productions

```
A = a A'
A' = b1
A' = b2
A = c
```

Properties of Grammar Transformations

Preservation

- Preserves set of sentences
- Preserves set of trees
- Preserves tree structure

Systematic

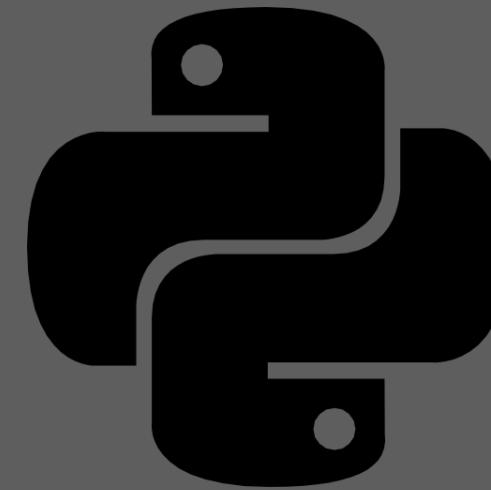
- Algorithmic
- Heuristic

Layout-Sensitive Syntax

Layout-Sensitive Syntax

```
if x ≠ y:  
    if x > 0:  
        y = x  
else:  
    y = -x
```

```
guessValue x = do  
    putStrLn "Enter your guess:"  
    guess ← getLine  
    case compare (read guess) x of  
        EQ → putStrLn "You won!"  
        _ → do putStrLn "Keep guessing."  
              guessValue x
```



Disambiguation with Indentation Sensitive Context-free Grammars

```
case → 'case'> exp= 'of'> altBlock=
  -- Reset indentation for delimited blocks

altBlock → '{'> alts* close*
close   → '}'>
  -- Increase indentation for non-delimited blocks

altBlock → altLayout>
altLayout → |alts≡ altLayout=
altLayout → |alts≡
  -- Clause sequences

alts → alt=
alts → alt= ';'> alts=
```

Michael Adams. Principled Parsing for Indentation-Sensitive Languages: Revisiting Landin's Offside Rule. In POPL'13.

SDF3: Disambiguation with Layout Constraints

context-free syntax

```
Impl          = Stm {layout(1.first.col < 1.left.col)}  
  
Impls.StmSeq = Impl Impl {layout(1.first.col = 2.first.col)}  
  
Impls        = Impl  
Expls        = Stm  
Expls.StmtSeq = Stm ";" Expls  
Stms.Stms    = Impls  
  
Stms.Stms    = "{" Expls "}" {ignore-layout}
```

Layout Constraints: Token Selectors

```
x = do 9 + 4
        * 3
main = do putStrLn $
        show (x *
              2) right
        last
```

first → `putStrLn $`
left → `show (x *`
right → `2)`
last → `last`

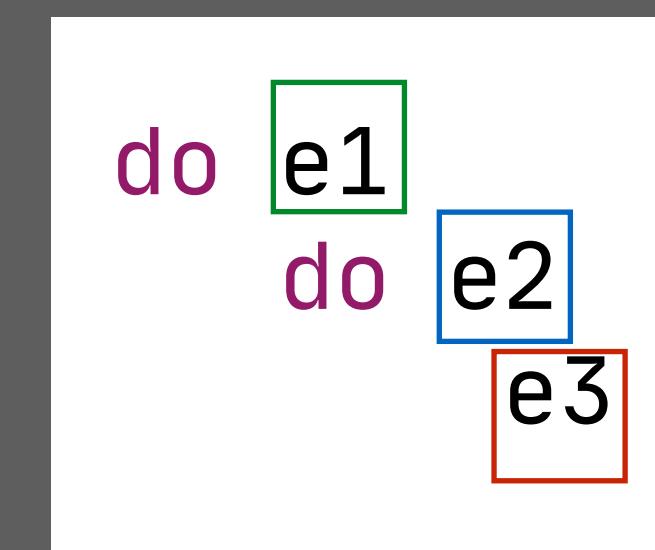
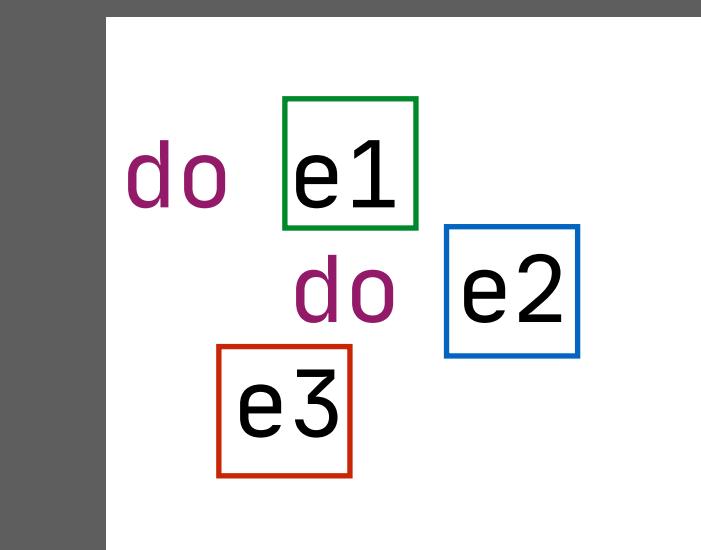
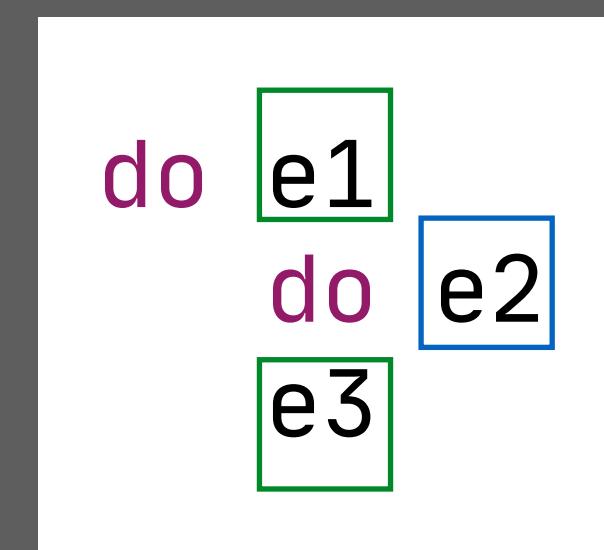
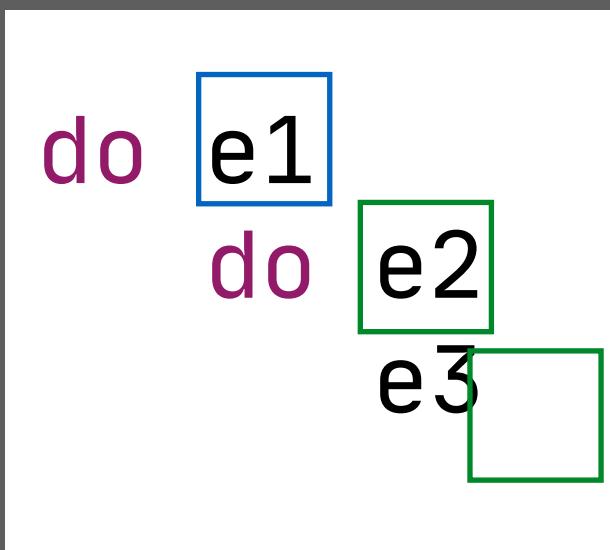
```
lComp = do
    x <- xRange
    return $ do
        y <- yRange
        return (x, y)
```

Interpret indentation using shapes around
the tokens that belong to a subtree.

Layout Constraints: Encoding Alignment

context-free syntax

```
Exp.Do      = "do" ExpList
ExpList.Cns = Exp
ExpList.Lst = ExpList Exp {layout(1.first.col = 2.first.col)}
Exp.Id      = ID
```



Layout Constraints: Encoding Indentation

```
Exp.IfElse = "if" Exp "then" Exp "else" Exp
{layout(4.first.line > 1.last.line && // then clause in next line
  4.first.col > 1.first.col && // and indented
  1.first.col == 5.first.col && // "if" and "else" aligned
  4.first.col == 6.first.col)} // then and else clauses aligned
```

```
if e1 then
  if e2 then
    e3
  else
    e4
else
  e5
```

Low-level declarations

Layout Declarations

Tree Selectors

context-free syntax

```
Exp.Do      = "do" ExpList
ExpList.Cns = Exp
ExpList.Lst = ExpList Exp {layout(1.first.col = 2.first.col)}
Exp.Id      = ID
```

context-free syntax

```
Exp.Do      = "do" ExpList
ExpList.Cns = Exp
ExpList.Lst = exps:ExpList exp:Exp {layout(... exps exp)}
Exp.Id      = ID
```

Tree Selectors

context-free syntax

```
Exp.Do      = "do" ExpList {layout(2.first.col > 1.first.col)}
ExpList.Cns = Exp
ExpList.Lst = ExpList Exp {layout(1.first.col = 2.first.col)}
Exp.Id     = ID
```

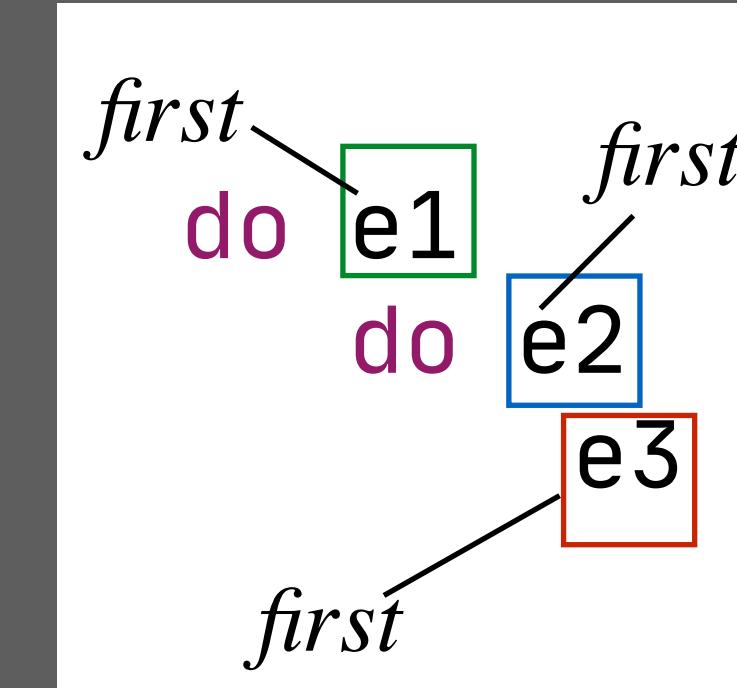
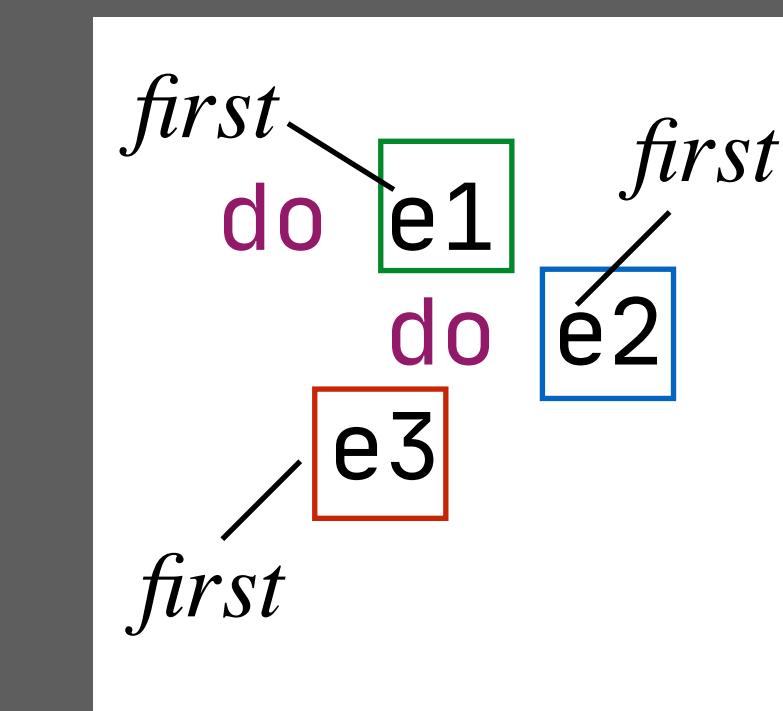
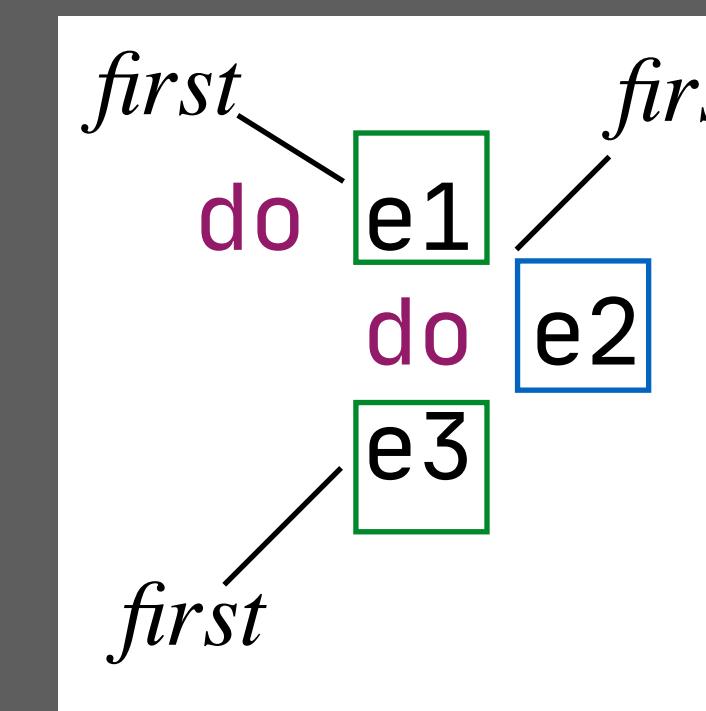
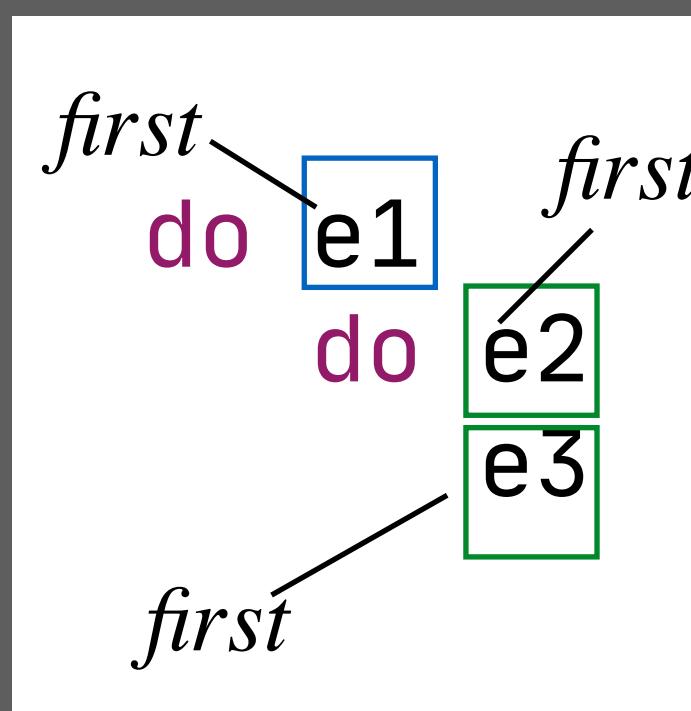
context-free syntax

```
Exp.Do      = "do" ExpList {layout(... "do" 1)}
ExpList.Cns = Exp
ExpList.Lst = exps:ExpList exp:Exp {layout(... exps exp)}
Exp.Id     = ID
```

Alignment with Layout Constraints

context-free syntax

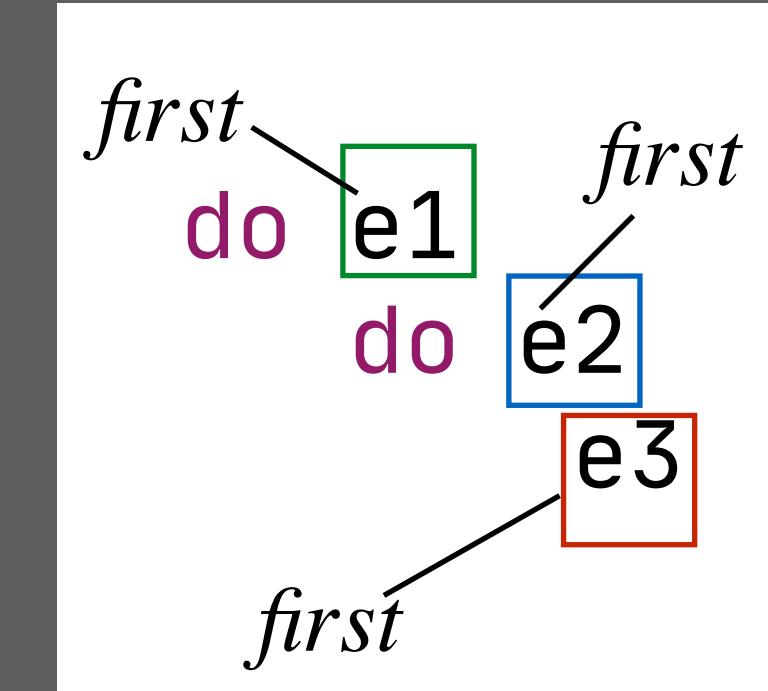
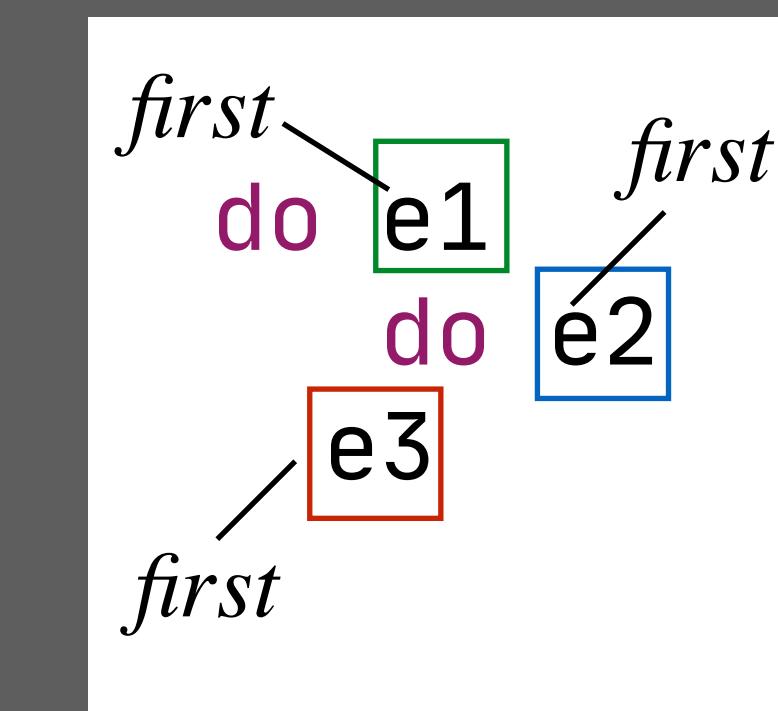
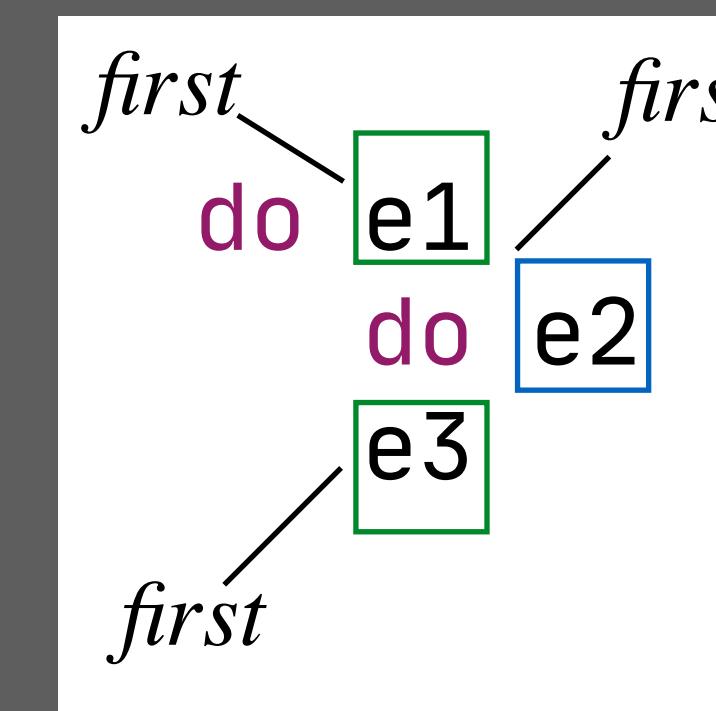
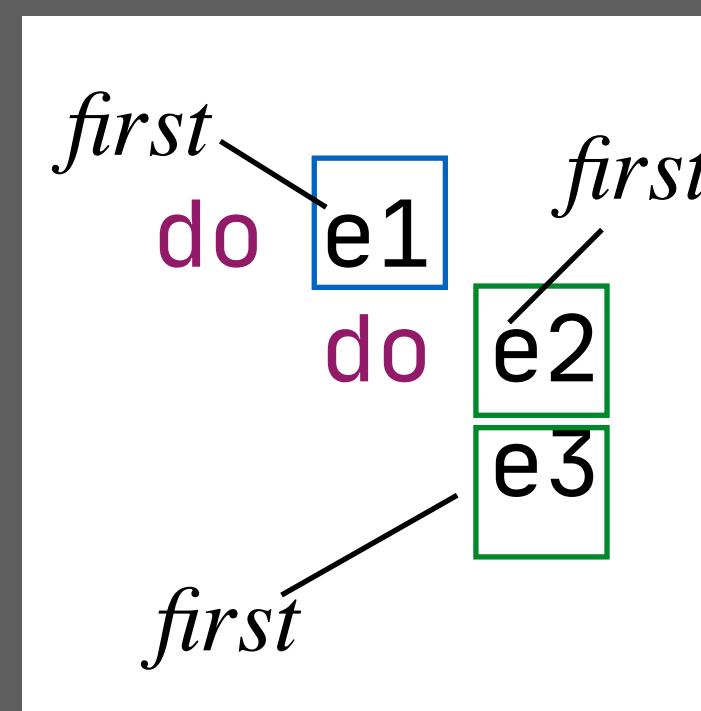
```
Exp.Do      = "do" ExpList
ExpList.Cns = Exp
ExpList.Lst = ExpList Exp {layout(1.first.col = 2.first.col)}
Exp.Id     = ID
```



Alignment Declaration

context-free syntax

```
Exp.Do      = "do" ExpList
ExpList.Cns = Exp
ExpList.Lst = exps:ExpList exp:Exp {layout(align exps exp)}
Exp.Id     = ID
```



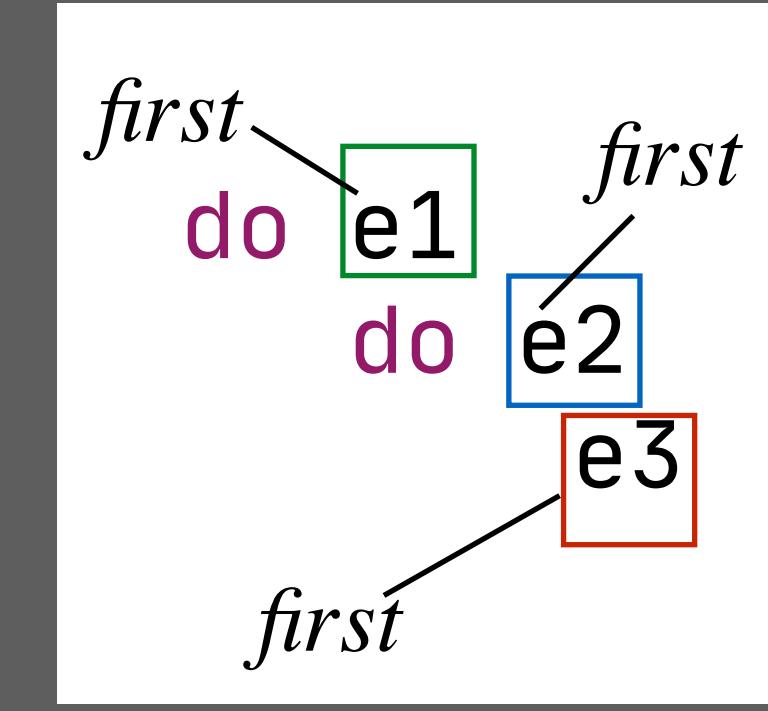
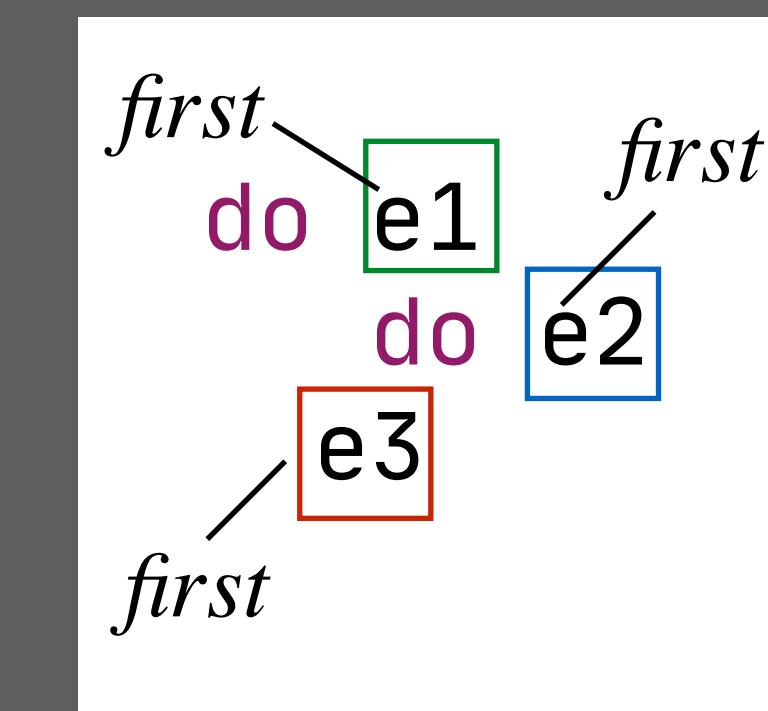
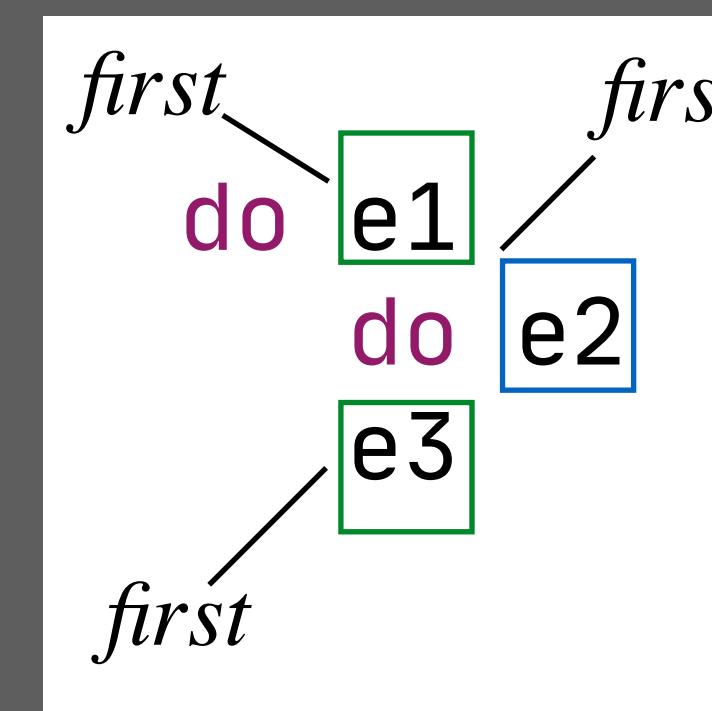
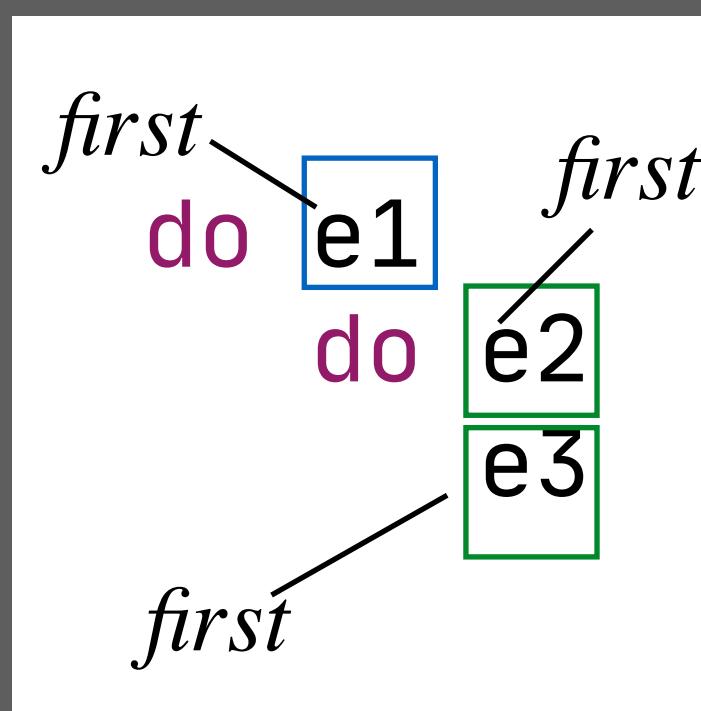
Semantics

$$\frac{x.\text{first.col} = y.\text{first.col}}{\text{align } x \text{ } y}$$

List Alignment Declaration

context-free syntax

```
Exp.Do  = "do" exps:Exp+ {layout(align-list exps)}  
Exp.Id  = ID  
Exp+    = Exp+ Exp // normalized  
Exp+    = Exp          // productions
```

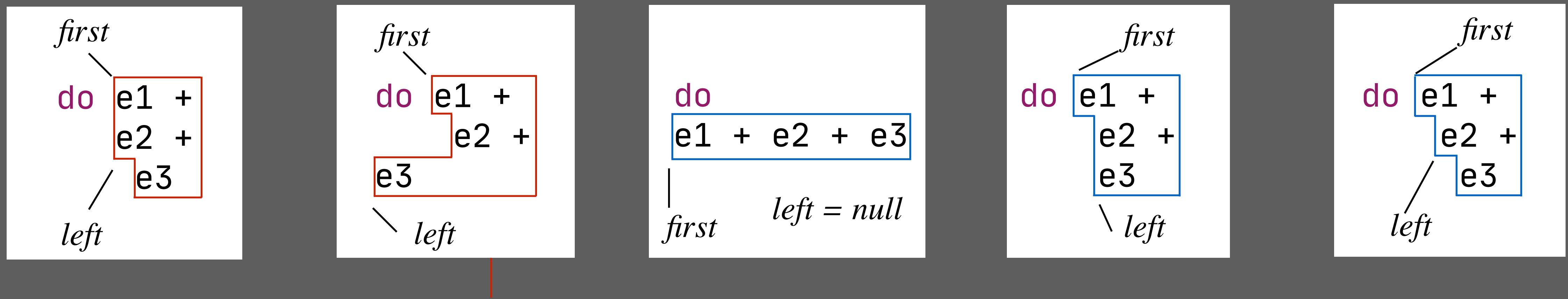


Semantics

$A^+ = A^+ \ A \ layout(1.\text{first}.col = 2.\text{first}.col)$

align-list x

Offside Rule



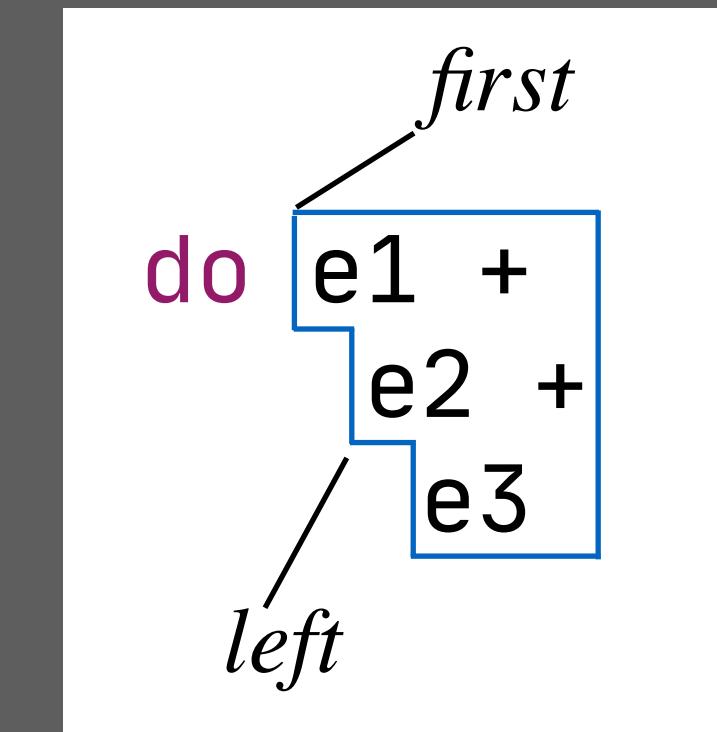
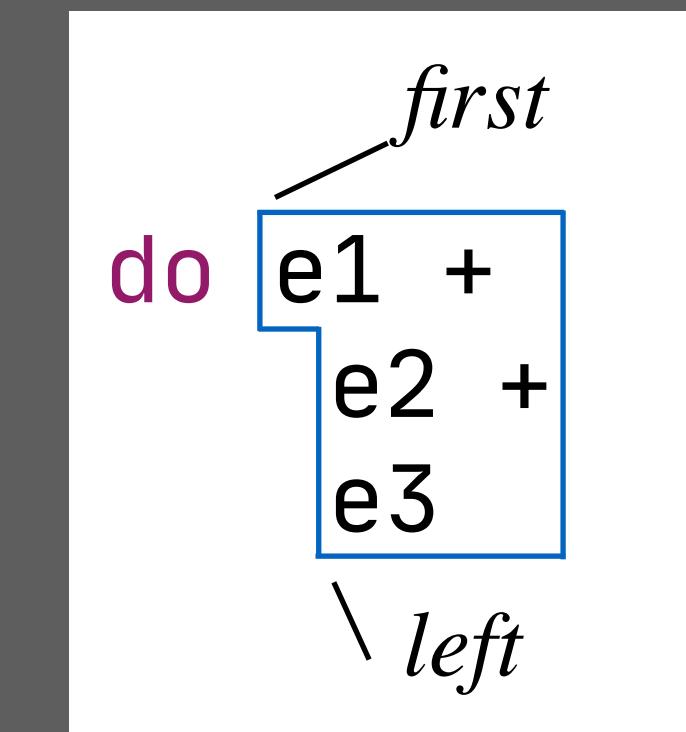
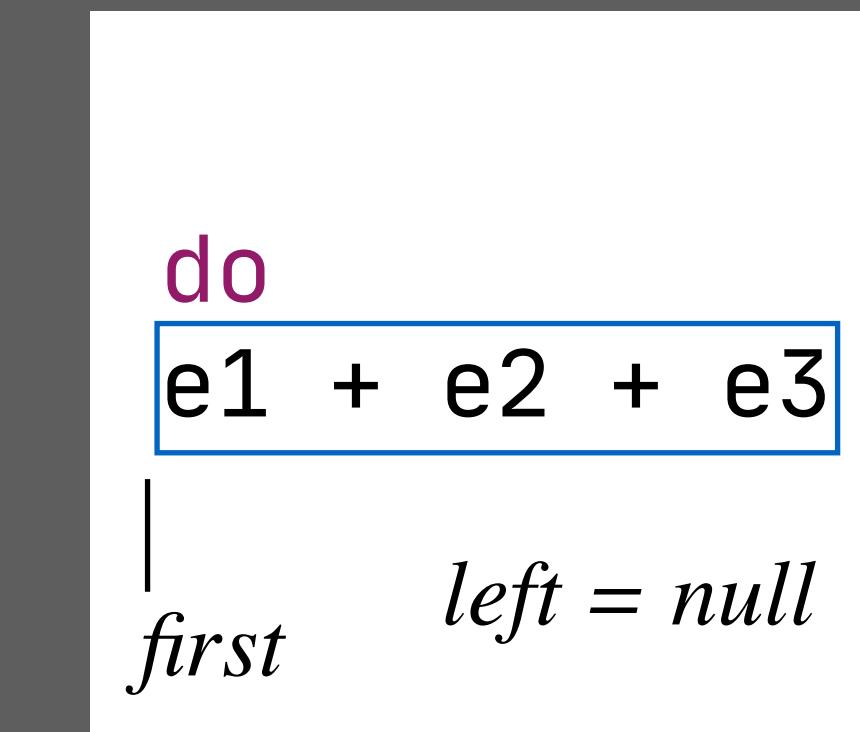
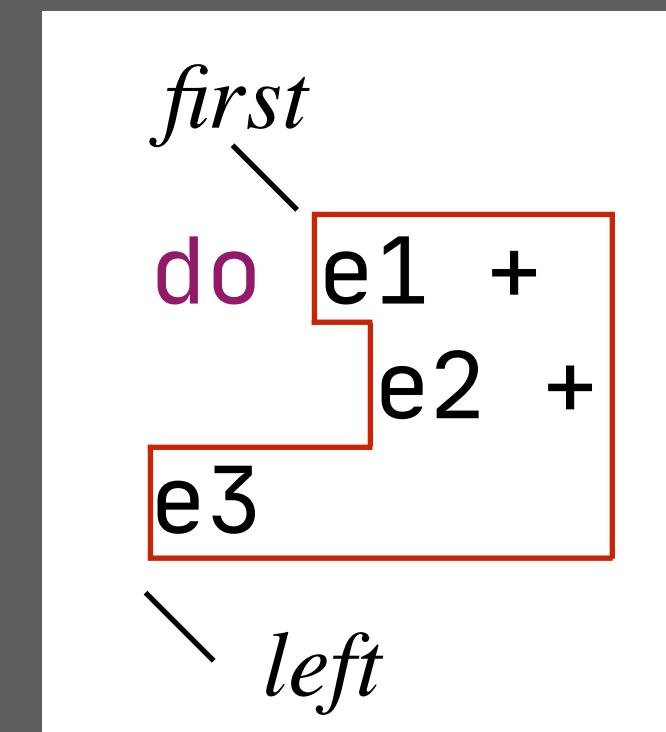
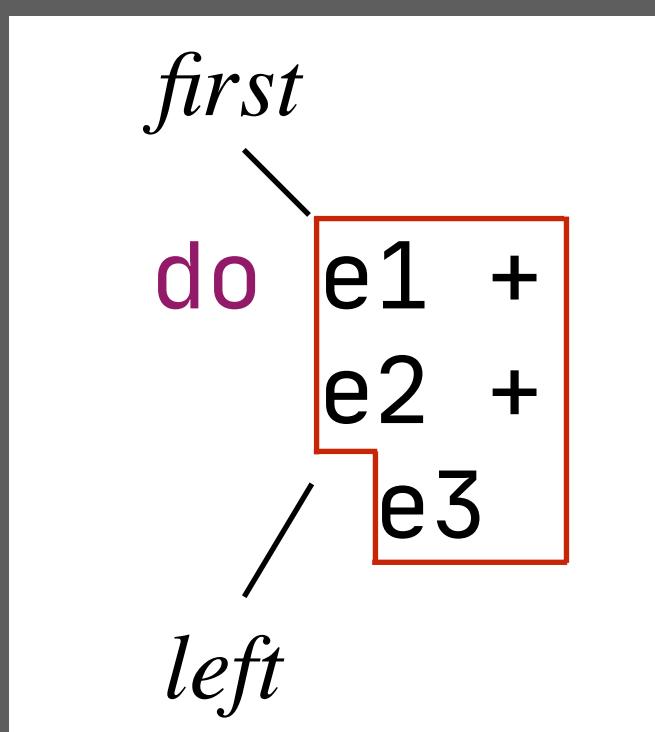
“The offside rule prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure.”

Erdweg et. al.. Layout-Sensitive Generalized Parsing. In SLE’12.

Offside with Layout Constraints

context-free syntax

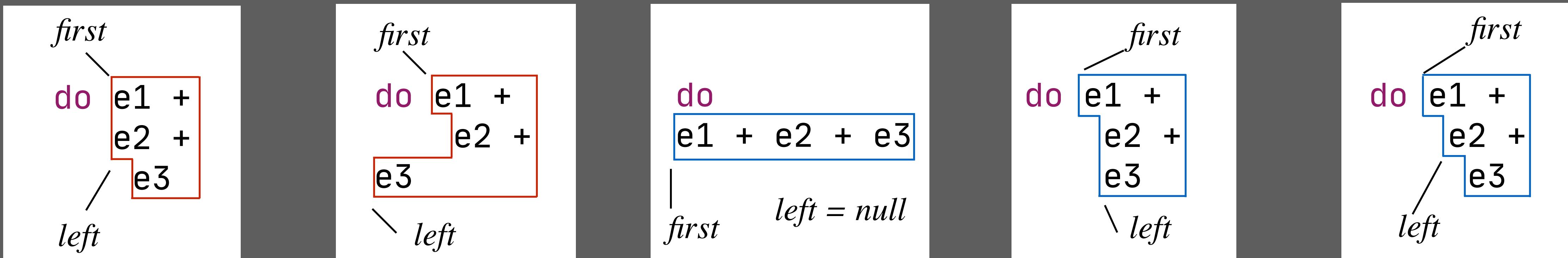
```
Exp.Do  = "do" Exp {layout(2.left.col > 2.first.col)}  
Exp.Add = Exp "+" Exp {left}  
Exp.Id  = ID
```



Offside

context-free syntax

```
Exp.Do  = "do" exp:Exp {layout(offside exp)}  
Exp.Add = Exp "+" Exp {left}  
Exp.Id  = ID
```



Semantics

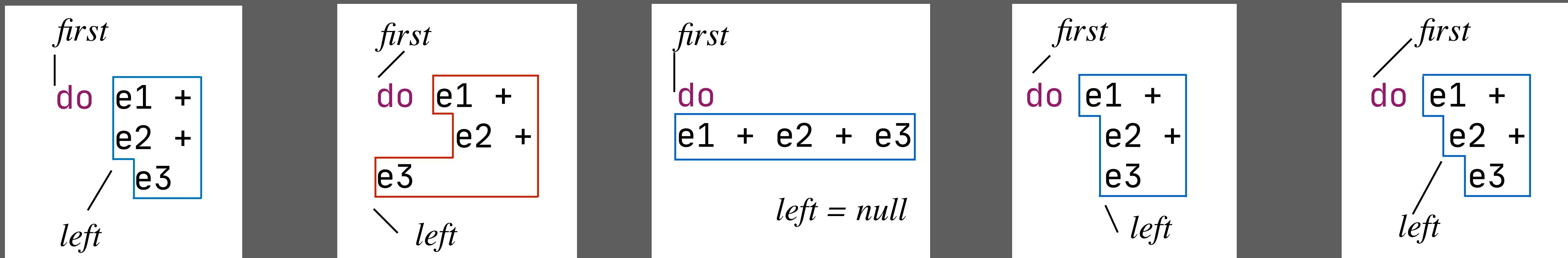
$x.left.col > x.first.col$

offside x

Relative Offside

context-free syntax

```
Exp.Do  = "do" exp:Exp {layout(offside "do" exp)}  
Exp.Add = Exp "+" Exp {left}  
Exp.Id  = ID
```



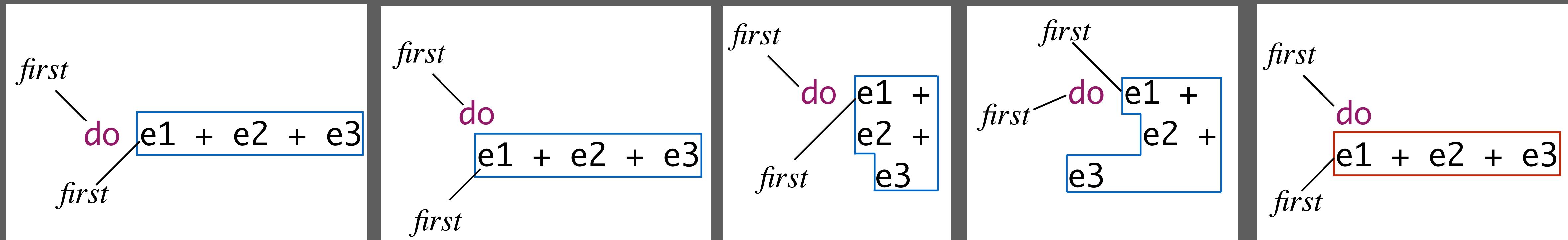
Semantics

$$\frac{y.\text{left}.col > x.\text{first}.col}{\text{offside } x \ y}$$

Indentation

context-free syntax

```
Exp.Do  = "do" exp:Exp {layout(indent "do" exp)}  
Exp.Add = Exp "+" Exp {left}  
Exp.Id  = ID
```



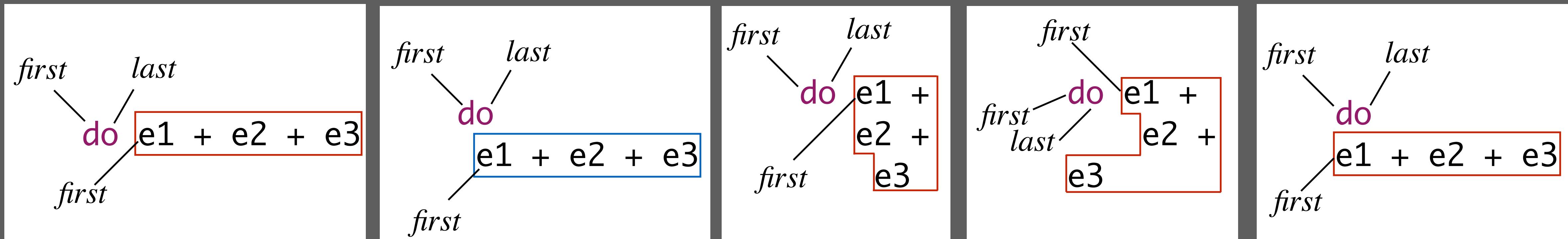
Semantics

$$\frac{y.\text{first.col} > x.\text{first.col}}{\text{indent } x \ y}$$

Newline + Indentation

context-free syntax

```
Exp.Do  = "do" exp:Exp {layout(newline-indent "do" exp)}  
Exp.Add = Exp "+" Exp {left}  
Exp.Id  = ID
```



Semantics

$y.\text{first.col} > x.\text{first.col} \& \& y.\text{first.line} > x.\text{last.line}$
newline-indent $x \cdot y$

Parsing and Pretty-Printing

{S} spoofax

The diagram illustrates the process of transforming source code into an abstract syntax tree (AST) and then into pretty-printed output. It consists of four windows arranged vertically:

- LayoutSens.sdf3**: Shows the SDF grammar definition for the LayoutSens module, defining rules for context-free syntax, priorities, and lexical syntax.
- example1.lsn**: Shows the input source code:

```
1 if e1 then if e2 then e3
2           else e4
3 else      e5
4
5 do x = 1
6   y = 2
7   x
8   + y
```
- example1.aterm**: Shows the resulting Abstract Syntax Tree (AST) in ATERM format, generated by the parser:

```
1 Exps(
2   [ IfElse("e1", IfElse("e2", "e3", "e4"))
3     , Do([Assign("x", "1"), Assign("y", "2")
4       , Add("x", "y")]
5   )
6 )
```
- example1.pp.lsn**: Shows the pretty-printed version of the source code:

```
1 if e1 then
2   if e2 then
3     e3
4   else
5     e4
6 else
7   e5
8
9 do x = 1
10  y = 2
11
12 x + y
```

Two large blue arrows indicate the flow: a downward-pointing arrow labeled "Parse" between the second and third windows, and another downward-pointing arrow labeled "Pretty-Print" between the third and fourth windows.

Bottom status bar: Writable Insert 1 : 1

Reading Material

Introduces layout constraints

https://doi.org/10.1007/978-3-642-36089-3_14

Layout-Sensitive Generalized Parsing

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann

University of Marburg, Germany

Abstract. The theory of context-free languages is well-understood and context-free parsers can be used as off-the-shelf tools in practice. In particular, to use a context-free parser framework, a user does not need to understand its internals but can specify a language *declaratively* as a grammar. However, many languages in practice are not context-free. One particularly important class of such languages is layout-sensitive languages, in which the structure of code depends on indentation and whitespace. For example, Python, Haskell, F#, and Markdown use indentation instead of curly braces to determine the block structure of code. Their parsers (and lexers) are not declaratively specified but hand-tuned to account for layout-sensitivity.

To support *declarative* specifications of layout-sensitive languages, we propose a parsing framework in which a user can annotate layout in a grammar. Annotations take the form of constraints on the relative positioning of tokens in the parsed subtrees. For example, a user can declare that a block consists of statements that all start on the same column. We have integrated layout constraints into SDF and implemented a layout-sensitive generalized parser as an extension of generalized LR parsing. We evaluate the correctness and performance of our parser by parsing 33 290 open-source Haskell files. Layout-sensitive generalized parsing is easy to use, and its performance overhead compared to layout-insensitive parsing is small enough for practical application.

1 Introduction

Most computer languages prescribe a textual syntax. A parser translates from such textual representation into a structured one and constitutes the first step in processing a document. Due to the development of parser frameworks such as lex/yacc [15], ANTLR [18,17], PEGs [6,7], parsec [13], or SDF [8], parsers can be considered off-the-shelf tools nowadays: Non-experts can use parsers, because language specifications are declarative. Although many parser frameworks support some form of context-sensitive parsing (such as via semantic predicates in ANTLR [18]), one particularly relevant class of languages is not supported declaratively by any existing parser framework: layout-sensitive languages.

Layout-sensitive languages were proposed by Landin in 1966 [12]. In layout-sensitive languages, the translation from a textual representation to a structural one depends on the code's layout and its indentation. Most prominently, the *off-side rule* prescribes that all non-whitespace tokens of a structure must be further to the right than the token that starts the structure. In other words, a token

Introduces layout declarations to abstract from low-level layout constraints and pretty-printing based on layout declarations/constraints.

Includes summary of layout constraints

Won distinguished paper award at SLE'18.

<https://doi.org/10.1145/3276604.3276607>



Declarative Specification of Indentation Rules

A Tooling Perspective on Parsing and Pretty-Printing Layout-Sensitive Languages

Luís Eduardo de Souza Amorim
Delft University of Technology
The Netherlands
l.e.desouzaamorim-1@tudelft.nl

Sebastian Erdweg
Delft University of Technology
The Netherlands
s.t.erdweg@tudelft.nl

Michael J. Steindorfer
Delft University of Technology
The Netherlands
michael@steindorfer.name

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

Abstract

In layout-sensitive languages, the indentation of an expression or statement can influence how a program is parsed. While some of these languages (e.g., Haskell and Python) have been widely adopted, there is little support for software language engineers in building tools for layout-sensitive languages. As a result, parsers, pretty-printers, program analyses, and refactoring tools often need to be handwritten, which decreases the maintainability and extensibility of these tools. Even state-of-the-art language workbenches have little support for layout-sensitive languages, restricting the development and prototyping of such languages.

In this paper, we introduce a novel approach to declarative specification of layout-sensitive languages using *layout declarations*. Layout declarations are high-level specifications of indentation rules that abstract from low-level technicalities. We show how to derive an efficient layout-sensitive generalized parser and a corresponding pretty-printer automatically from a language specification with layout declarations. We validate our approach in a case-study using a syntax definition for the Haskell programming language, investigating the performance of the generated parser and the correctness of the generated pretty-printer against 22191 Haskell files.

CCS Concepts • Software and its engineering → Syntax; Parsers;

Keywords parsing, pretty-printing, layout-sensitivity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. *SLE '18, November 5–6, 2018, Boston, MA, USA*

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6029-6/18/11...\$15.00
<https://doi.org/10.1145/3276604.3276607>

ACM Reference Format:
Luís Eduardo de Souza Amorim, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018. Declarative Specification of Indentation Rules: A Tooling Perspective on Parsing and Pretty-Printing Layout-Sensitive Languages. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3276604.3276607>

1 Introduction

Layout-sensitive (also known as indentation-sensitive) languages were introduced by Landin [17]. The term characterizes languages that must obey certain *indentation rules*, i.e., languages in which the indentation of the code influences how the program should be parsed. In layout-sensitive languages, alignment and indentation are essential to correctly identify the structures of a program. Many modern programming languages including Haskell [11], Python [23], Markdown [14] and YAML [4] are layout-sensitive. To illustrate how layout can influence parsing programs in such languages, consider the Haskell program in Figure 1, which contains multiple *do*-expressions:

```
1 guessValue x = do
2   putStrLn "Enter your guess:"
3   guess <- getLine
4   case compare (read guess) x of
5     EQ -> putStrLn "You won!"
6     _ -> do putStrLn "Keep guessing."
7   guessValue x
```

Figure 1. *Do*-expressions in Haskell.

In Haskell, all statements inside a *do*-block should be aligned (i.e., should start at the same column). In Figure 1, we know that the statement on line 7 (*guessValue x*) belongs to the inner *do*-block solely because of its indentation. If we modify the indentation of this statement, aligning it with the statements in the outer *do*-block, the program would have a different interpretation, looping indefinitely.

Next: Static Analysis and Type Checking