Declarative Syntax Definition with SDF3

Eelco Visser

TU Delft

PLDI | ‘London’ | June 16, 2020

Declarative Syntax Definition with SDF3

Eelco Visser

Joint work with Eduardo Amorim, Jasper Denkers, Sebastian Erdweg, Lennart Kats, Maartje de Jonge, Tobi Vollebregt, …

Goals of SDF3 Design

Syntax definition

- Define concrete and abstract syntax of programming languages

Understandable

- Can be used as reference documentation

Executable

- Can be used to generate tools

Declarative

- No need to understand (parsing) algorithms

Multi-purpose

- Derive many/all syntactic services from single definition

A Work in Progress

SDF

- Heering, Hendriks, Klint, Rekers 1989
- Generalized-LR parsing

SDF2

- Visser 1997
- Scannerless Generalized-LR parsing
- Shallow priority conflicts in LR table

SDF3

- Amorim, Visser, and many others (since 2009)
- Deep priority conflicts
- Layout-sensitive syntax
- Constructors, templates, completion, …

SDF3 in Propositions

Basic language design is simple

- Core = context-free grammars
- Boilerplate to define all aspects of language syntax

SDF3 provides high-level sugar

- Convenient, concise expression
- Abstracts from boilerplate

Hidden design

- Surface level is deceptively simple
- Mostly ‘does what you expect’

This talk: Explain these by means of propositions

- E.g. “Syntax = Structure”

SDF3 in Propositions

Syntax = Structure

Lexical Syntax ~= Context-Free Syntax

Parsing = Formatting^{-1}

Completion = Rewriting Incomplete Sentences

Disambiguation = Choosing Structure

Parenthesize = Disambiguate^{-1}

Parse Error Recovery = Parsing with Permissive Grammar

Reserved words = reject

Prefer longest match = follow restrictions

Layout-sensitive syntax = context-free syntax + 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
```

```
Let(
  [ 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

  "+"        = [\43]
  "function" = [\102] [\117] [\110] [\99]
               [\116] [\105] [\111] [\110]
  "{"        = [\123]
  "}"        = [\125]
  "("        = [\40]
  ","        = [\44]
  ")"        = [\41]
  "let"      = [\108] [\101] [\116]
  "in"       = [\105] [\110]
  "end"      = [\101] [\110] [\100]
  "="        = [\61]
```

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 ")"
```

Parsing = Formatting^{-1}

Parsing = Formatting^{-1}

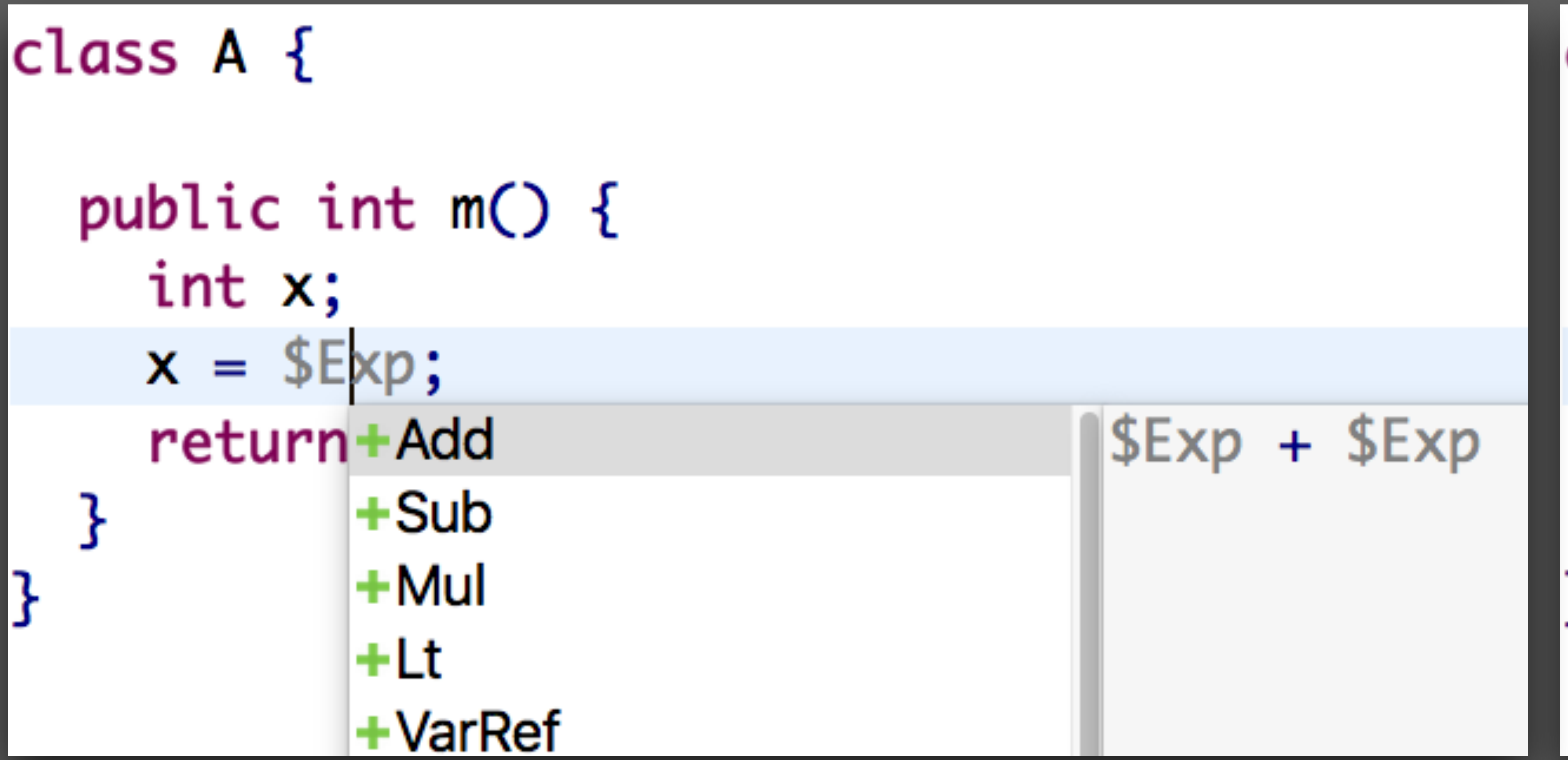
```
context-free syntax

  Exp.Var = <<ID>>

  Exp.Int = <<INT>>

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

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

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

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

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

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

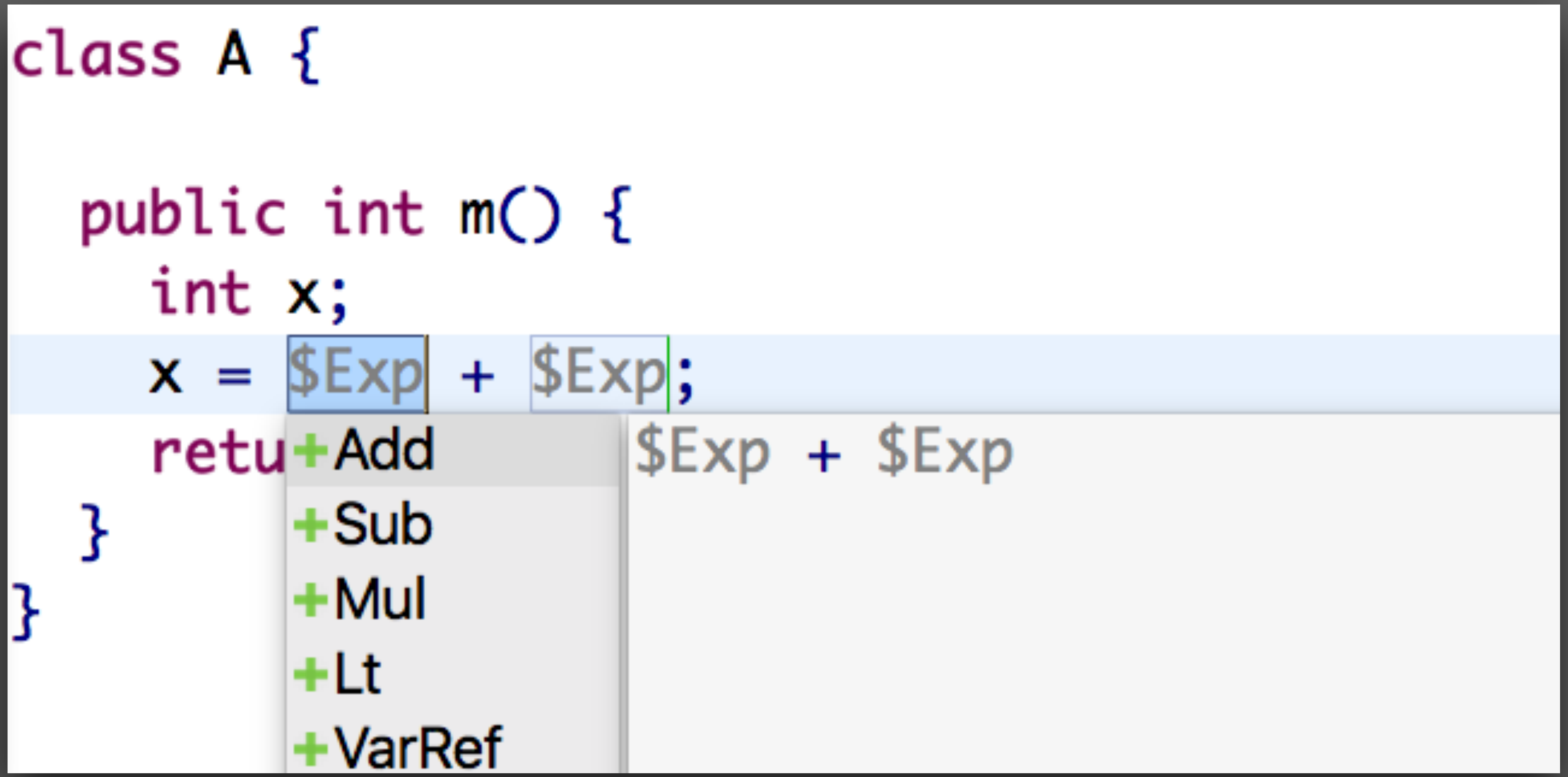
```
Let(
  [ Bnd(
      "inc"
    , Fun(["x"], Add(Var("x"), Int("1")))
    )
  ]
, App(Var("inc"), [Int("3")])
)
```

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

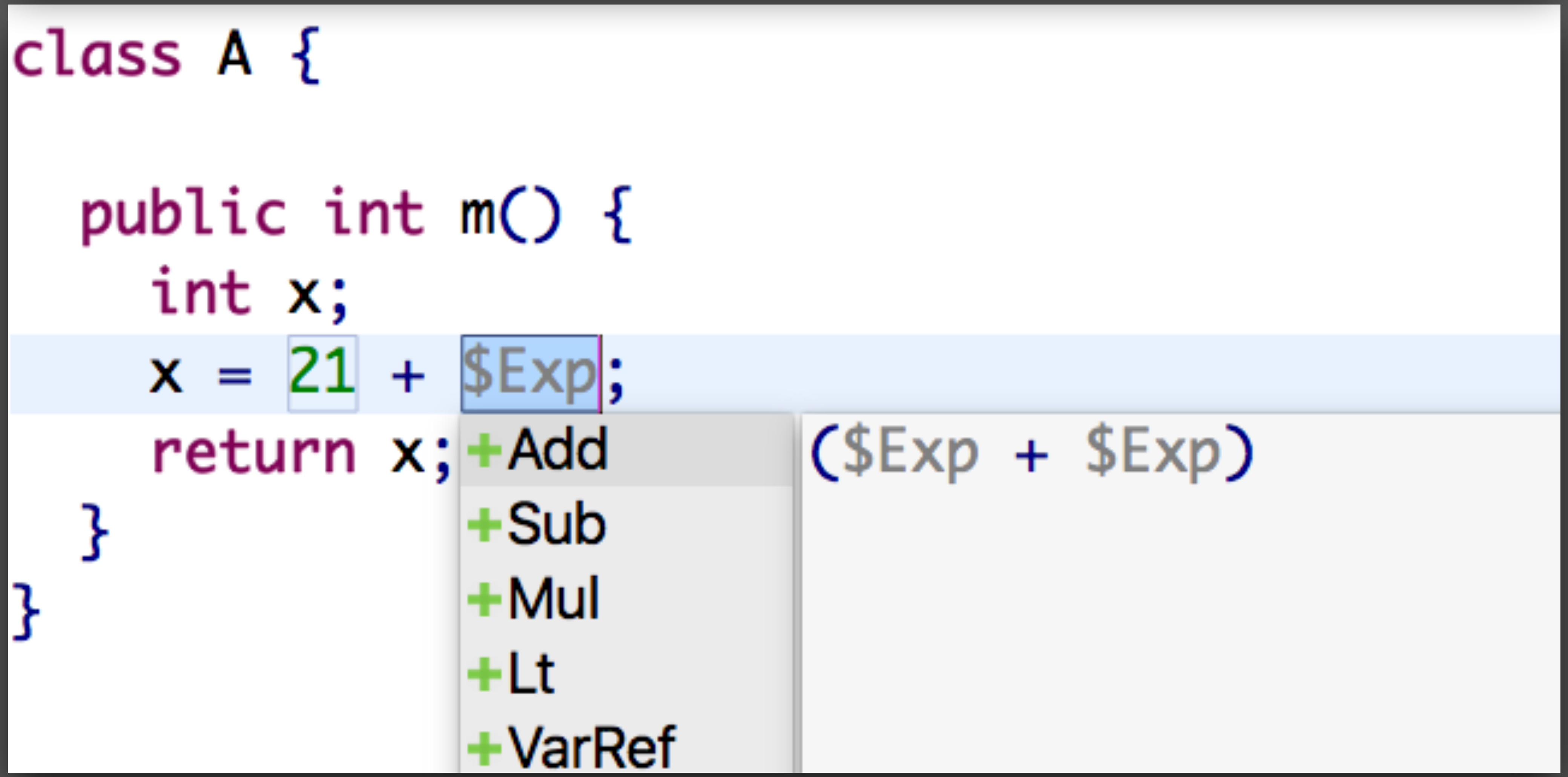
Completion = Rewrite(Incomplete Structure)



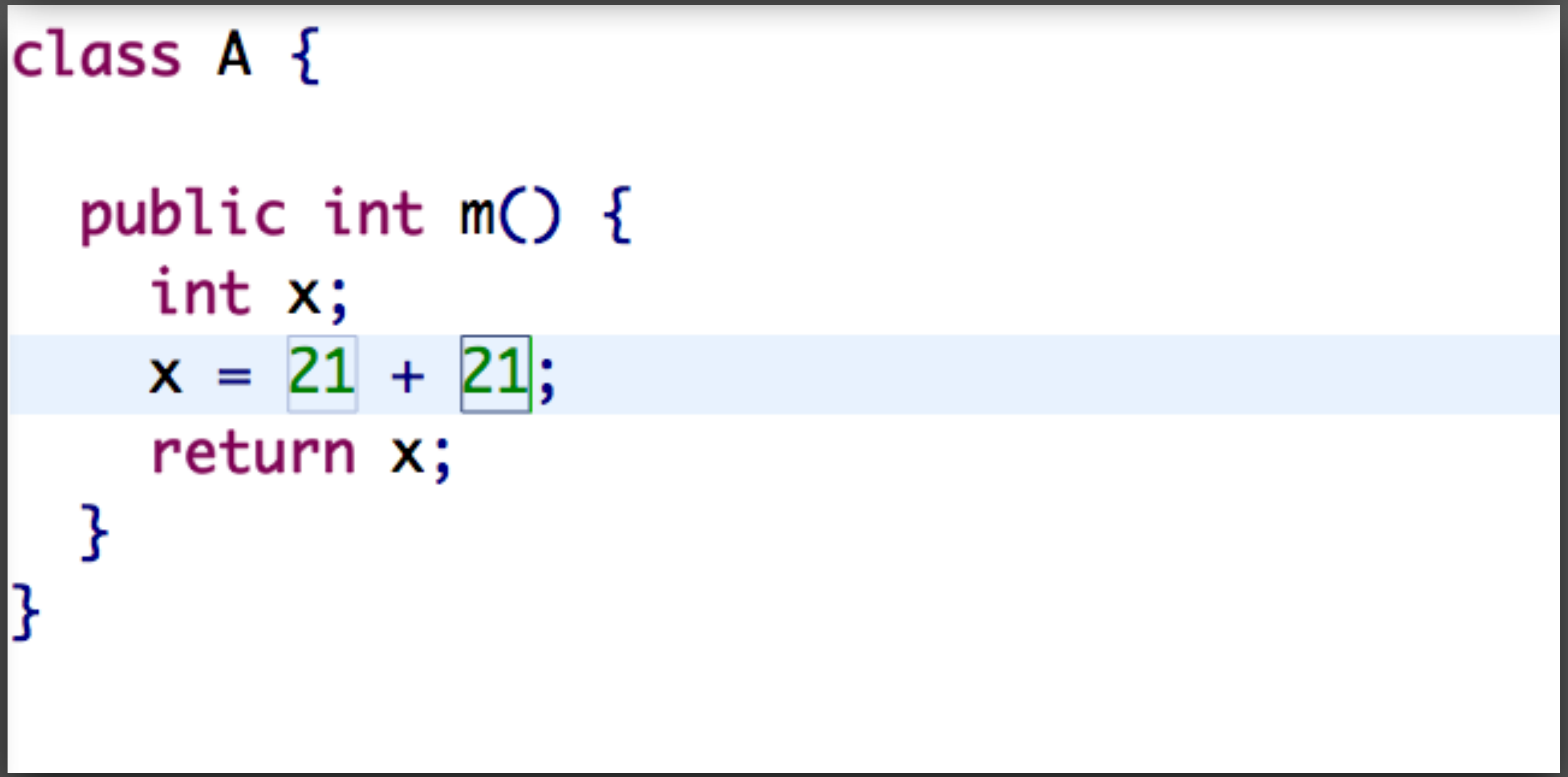
```
class A {

  public int m() {
    int x;
    x = $Exp;
    return
  }
}
```



```
class A {

  public int m() {
    int x;
    x = $Exp + $Exp;
    retu
  }
}
```



```
class A {

  public int m() {
    int x;
    x = 21 + $Exp;
    return x;
  }
}
```



```
class A {

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

Extension

Language Extension => Grammar Extension

```
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"))
)
```

Disambiguation

Use tree pictures for 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>>
```

$\Rightarrow$

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
```

$\Downarrow$

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

$\Downarrow$

```
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
```

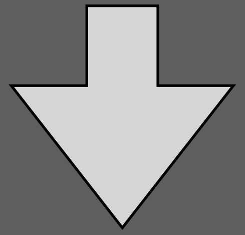
$\Downarrow$

```
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))
```

$\Downarrow$

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

$\Downarrow$

```
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
```