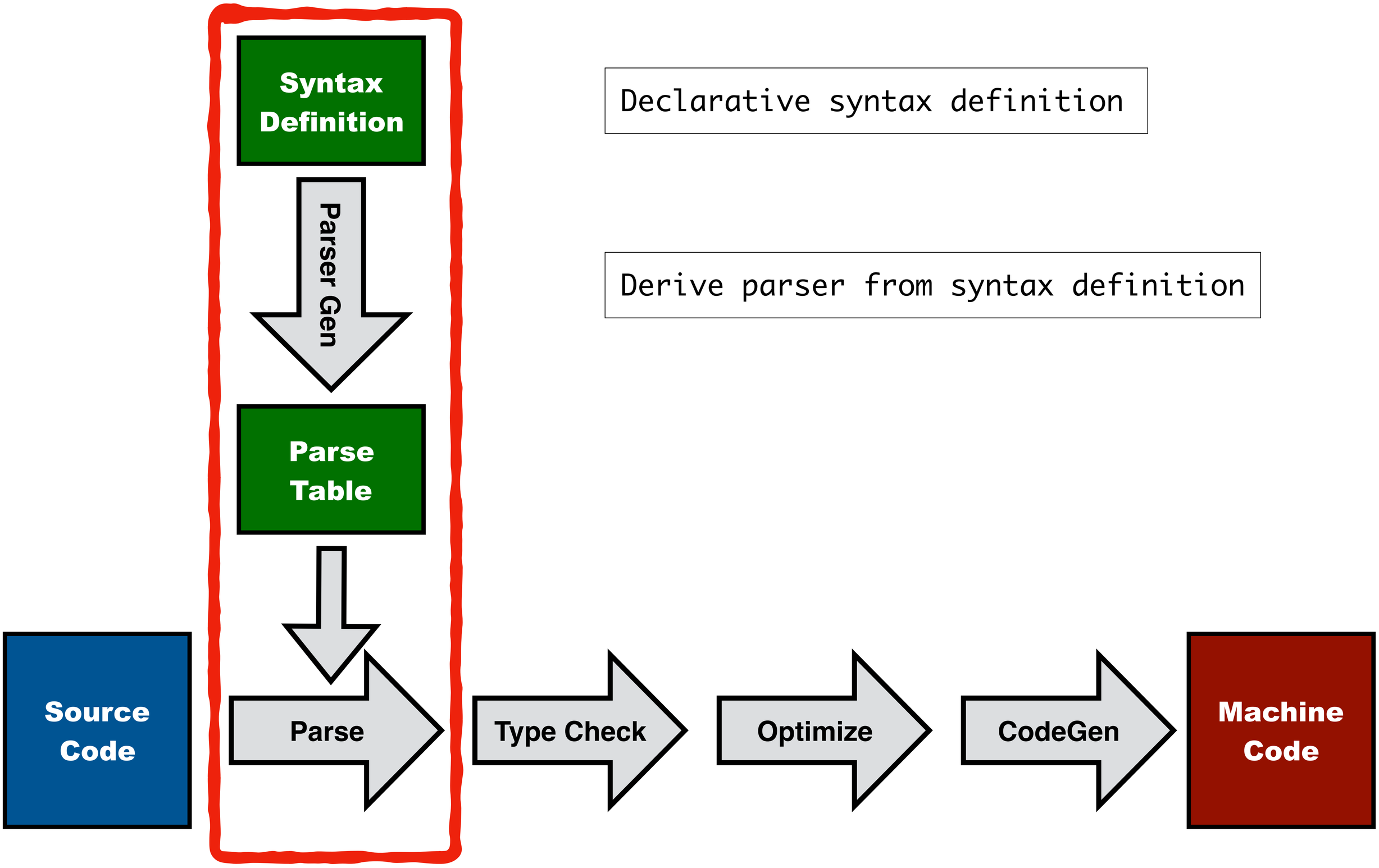
Declare Your Language

Chapter 3: Syntactic (Editor) Services

Eelco Visser

IN4303 Compiler Construction
TU Delft
September 2017

TU Delft



Declarative syntax definition

Derive parser from syntax definition

This Lecture

Lexical syntax

- defining the syntax of tokens / terminals including layout
- making lexical syntax explicit

Formatting specification

- how to map (abstract syntax) trees to text

Syntactic completion

- proposing valid syntactic completions in an editor

~~**Grammar transformation**~~

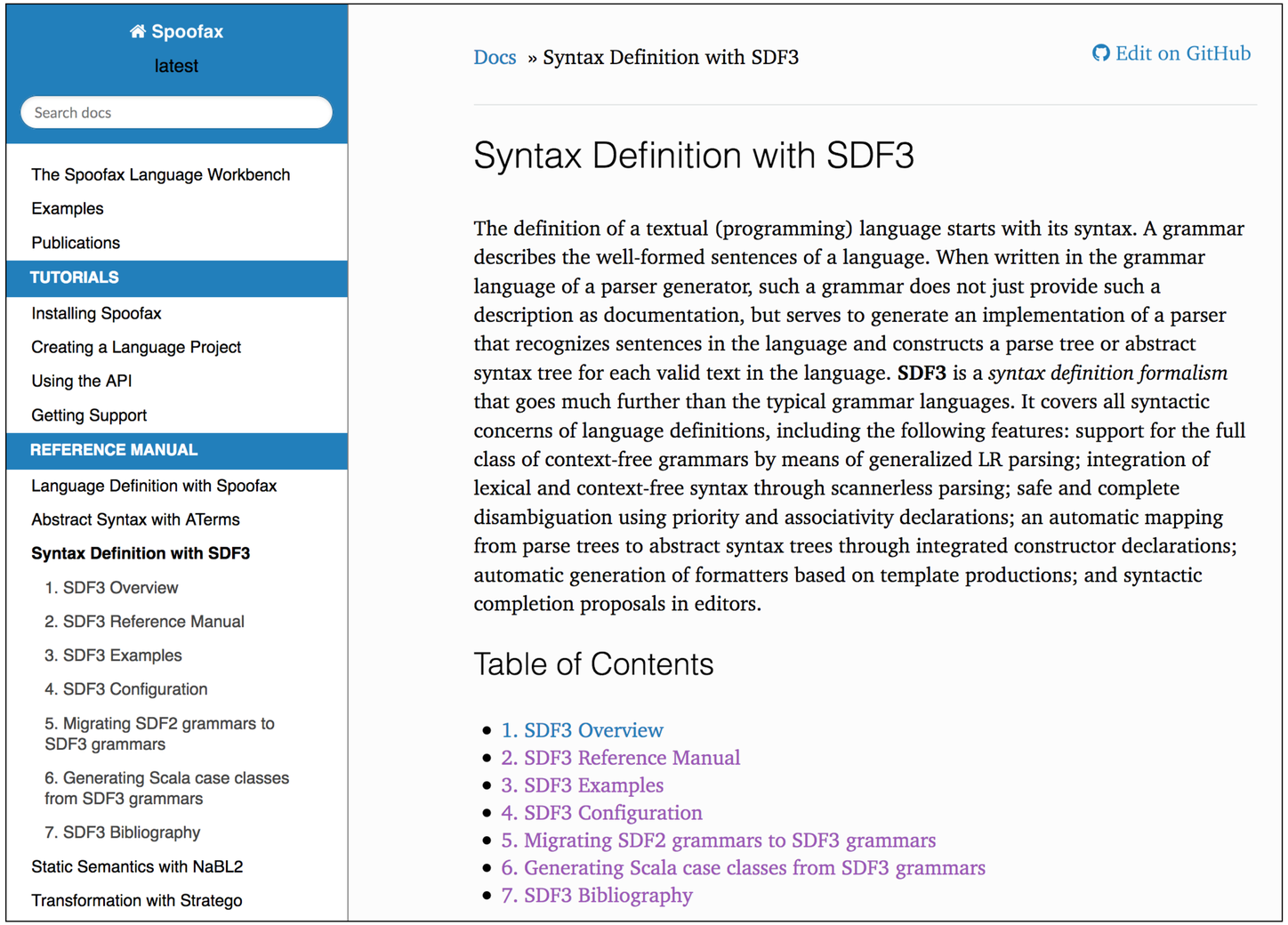
- ~~disambiguation by transformation~~

~~**Parsing**~~

- ~~interpreting a syntax definition to map text to trees~~

Reading Material

The SDF3 syntax definition formalism is documented at the metaborg.org website.



http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/index.html

The inverse of parsing is unparsing or pretty-printing or formatting, i.e. mapping a tree representation of a program to a textual representation. A plain context-free grammar can be used as specification of an unparser. However, then it is unclear where the whitespace should go.

This paper extends context-free grammars with templates that provide hints for layout of program text when formatting.

https://doi.org/10.1145/2427048.2427056

Declarative Specification of Template-Based Textual Editors

Tobi Vollebregt
tobivollebregt@gmail.com

Lennart C. L. Kats
l.c.l.kats@tudelft.nl

Eelco Visser
visser@acm.org

Software Engineering Research Group
Delft University of Technology
The Netherlands

ABSTRACT

Syntax discoverability has been a crucial advantage of structure editors for new users of a language. Despite this advantage, structure editors have not been widely adopted. Based on immediate parsing and analyses, modern textual code editors are also increasingly syntax-aware: structure and textual editors are converging into a new editing paradigm that combines text and templates. Current text-based language workbenches require redundant specification of the ingredients for a template-based editor, which is detrimental to the quality of syntactic completion, as consistency and completeness of the definition cannot be guaranteed.

In this paper we describe the design and implementation of a specification language for syntax definition based on templates. It unifies the specification of parsers, unparsers and template-based editors. We evaluate the template language by application to two domain-specific languages used for tax benefits and mobile applications.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*pretty printers, program editors*; D.3.1 [**Programming Languages**]: Processors—*parsing, code generation*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*syntax*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.11 [**Software Architectures**]: Languages

1. INTRODUCTION

Language-aware structure editors provide a template-based paradigm for editing programs. They allow composing programs by selecting a template and filling in the placeholders, which can again be extended using templates. A crucial advantage of structure editors is syntax discoverability, helping new users to learn a language by presenting possible syntactic completions in a menu. Structure editors can be automatically generated from a syntax definition. Notable projects aiming at automatic generation of structure editors include MPS [23] and the Intentional Domain Workbench [19]. Structure editors can be used for general-purpose languages or for domain-specific languages (DSLs). A modern example of the former is the structure editor in MPS [23], for extensible languages based on Java. An example of the latter category is the DSL for modeling tax-benefit rules developed by IT services company Capgemini using the Cheetah system. Cheetah's facilities for discoverability and the use of templates are particularly effective to aid a small audience of domain expert programmers manage the verbose syntax based on legal texts.

Despite their good support for discoverability, structure editors have not been widely adopted. Pure structure editors tend to introduce an increased learning curve for basic editing operations. For example, they only support copy-pasting operations that maintain well-formedness of the tree and require small, yet non-trivial "refactoring" operations for editing existing code, e.g. when converting an `if` statement to an `if-else` statement. They also lack integration with other tools and expose the user to vendor lock-in. Transferring code across tools requires a shared representation that is generally not available. With software engineering tools such as issue trackers, forums, search, and version control being based on text, a textual representation is preferable, but requires the use of a parser and a parseable language syntax. This forces tools based on structure editors to find new solutions to problems long solved in the text domain.

To alleviate the problems of structure editors, there has been a long history of hybrid structure editors that introduce textual editing features to structure editors [24, 18, 11]. Conversely, modern textual code editors such as those in Eclipse and Visual Studio are increasingly syntax-aware, based on parsers that run while a program is edited. Over time, they have acquired features ranging from code folding to syntactic completions allowing programmers to fill in textual templates. Indeed, structure and textual editors are converging into a new editing paradigm that combines text and templates.

In order to provide the advantages of text editing to the tax-benefit DSL of Capgemini, we converted the language from the Cheetah system, which uses a structure editor, to the parser-based Spoofax language workbench [10]. We quickly realized that it would be impossible for a user to write new models in such a verbose language in the textual editor of Spoofax, without accurate and complete syntax discovery. In syntax-aware text editors this discovery is provided in the form of syntactic completion. Accurate and complete syntactic completion depends critically on two features of a language workbench: first, the syntactic completion proposals presented to the user must be relevant and complete, and second, it must be feasible to create and maintain the specification necessary to make the editor aware of these completion proposals.

LDTA 2012 Tallinn, Estonia

Syntax definitions cannot be used just for parsing, but for many other operations. This paper shows how syntactic completion can be provided generically given a syntax definition.

https://doi.org/10.1145/2427048.2427056

Principled Syntactic Code Completion using Placeholders

Luís Eduardo de Souza Amorim Sebastian Erdweg Guido Wachsmuth Eelco Visser
Delft University of Technology, Netherlands
l.e.desouzaamorim-1@tudelft.nl, s.t.erdweg@tudelft.nl, guwac@acm.org, e.visser@tudelft.nl

Abstract

Principled syntactic code completion enables developers to change source code by inserting code templates, thus increasing developer efficiency and supporting language exploration. However, existing code completion systems are ad-hoc and neither complete nor sound. They are not complete and only provide few code templates for selected programming languages. They also are not sound and propose code templates that yield invalid programs when inserted. This paper presents a generic framework that automatically derives complete and sound syntactic code completion from the syntax definition of arbitrary languages. A key insight of our work is to provide an explicit syntactic representation for incomplete programs using placeholders. This enables us to address the following challenges for code completion separately: (i) completing incomplete programs by replacing placeholders with code templates, (ii) injecting placeholders into complete programs to make them incomplete, and (iii) introducing lexemes and placeholders into incorrect programs through error-recovery parsing to make them correct so we can apply one of the previous strategies. We formalize our framework and provide an implementation in Spoofax.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments

Keywords Code Completion, Language Workbenches, IDEs

1. Introduction

Code completion, also known as content completion or content assist, is an editor service that proposes and performs expansion of the program text. Code completion helps the programmer to avoid misspellings and acts as a guide to discover

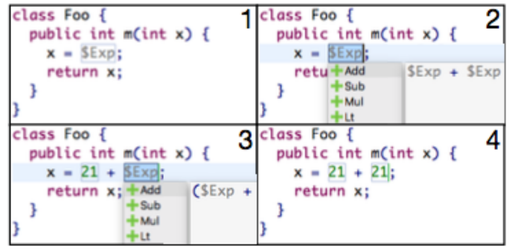


Figure 1. (1) Incomplete program with explicit placeholders. (2) Triggering completion for a placeholder. (3) After selecting a proposal, showing completions for nested placeholders. (4) Completing a nested placeholder by typing.

language features and APIs. Most mainstream integrated development environments (IDEs) provide some form of code completion and industrial studies indicate that code completion is one of the most frequently used IDE services [1].

There are two classes of code completion: syntactic and semantic. Syntactic code completion considers the syntactic context at the cursor position and proposes code templates for syntactic structures of the language. For example, most IDEs for Java support syntactic code completion with class and method templates. Semantic code completion also uses the cursor position to propose templates, but by applying semantic analysis to the program, the IDE can propose code templates or identifiers that do not violate the language's name binding or typing rules. For example, in this case IDEs for Java may suggest variables or methods that are visible in the current scope and have the expected type at the cursor position.

In this paper, we focus on *syntactic code completion*. Even for mainstream languages in mainstream IDEs, syntactic code completion is often ad-hoc and unreliable. Specifically, most existing services for syntactic code completion are incomplete and only propose code templates for selected language constructs of a few supported languages, thus inhibiting exploring the language's syntax. Moreover, most existing services are unsound and propose code templates that yield syntax errors when inserted at the cursor position.

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4447-0/16/10...$15.00
http://dx.doi.org/10.1145/2997364.2997374

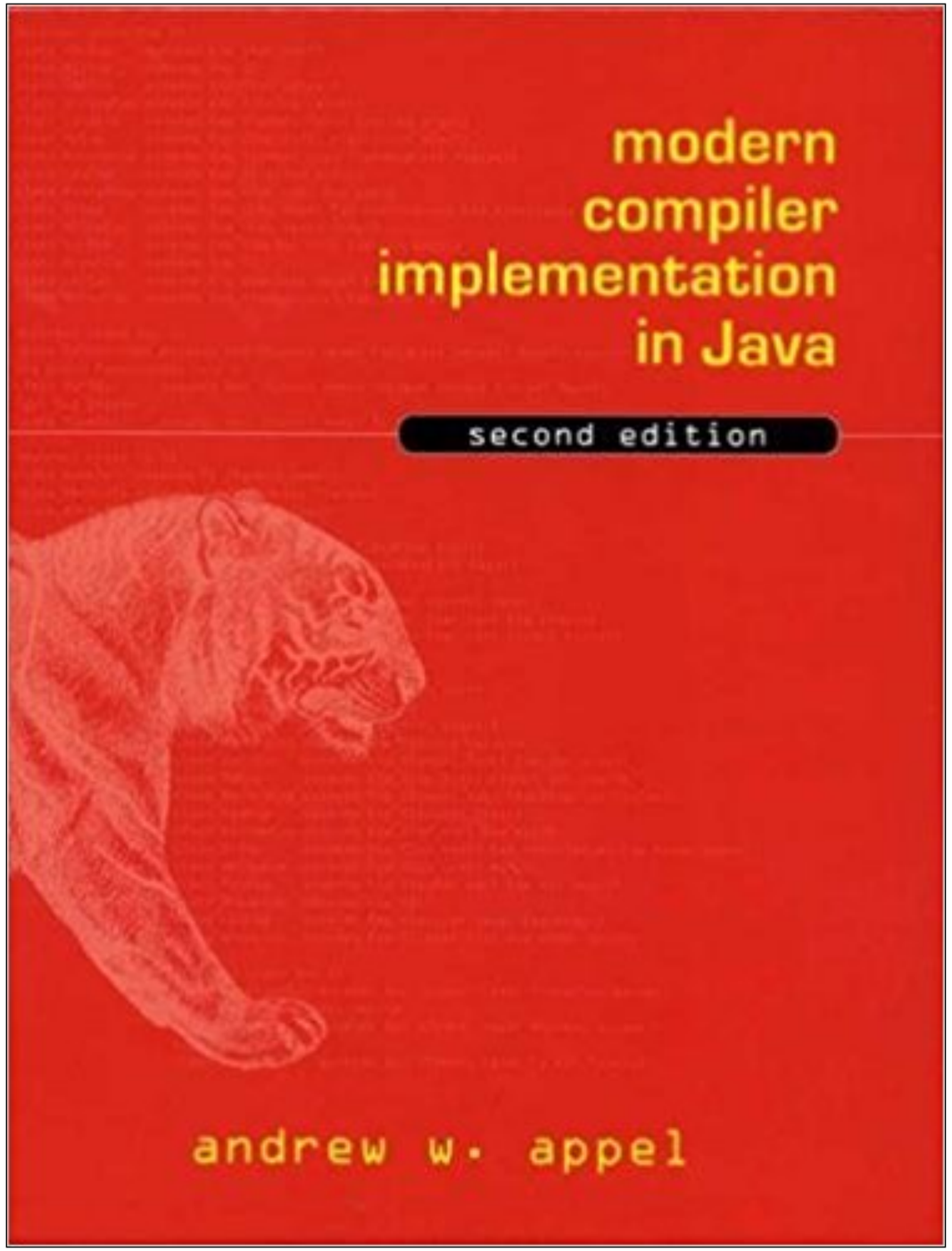
open access!

Chapter 2: Lexical Analysis

Chapter 3: Parsing

Chapter 4: Abstract Syntax

Next week: LR Parsing



Lexical Syntax

Context-Free Syntax vs Lexical Syntax

```
let function power(x: int, n: int): int =
      if n <= 0 then 1
      else x * power(x, n - 1)
 in power(3, 10)
end
```

phrase structure

```
Mod(
  Let(
    [ FunDecs(
        [ FunDec(
            "power"
          , [FArg("x", Tid("int")), FArg("n", Tid("int"))]
          , Tid("int")
          , If(
              Leq(Var("n"), Int("0"))
            , Int("1")
            , Times(
                Var("x")
              , Call(
                  "power"
                , [Var("x"), Minus(Var("n"), Int("1"))]
                )
              )
            )
          )
        ]
      )
    ]
  , [Call("power", [Int("3"), Int("10")])]
  )
)
```

separated by layout

lexeme / token

not separated by layout

structure not relevant

Character Classes

```
lexical syntax // character codes

  Character  = [\65]

  Range      = [\65-\90]

  Union      = [\65-\90] \/ [\97-\122]

  Difference = [\0-\127] / [\10\13]

  Union      = [\0-\9\11-\12\14-\255]
```

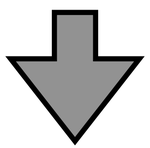
Character class represents choice from a set of characters

Sugar for Character Classes

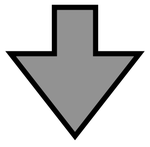
```
lexical syntax // sugar

  CharSugar  = [a]
             = [\97]

  CharClass  = [abcdefghijklmnopqrstuvwxyz]
             = [\97-\122]

  SugarRange = [a-z]
             = [\97-\122]

  Union      = [a-z] \/ [A-Z] \/ [0-9]
             = [\48-\57\65-\90\97-\122]

  RangeCombi = [a-z0-9\_]
             = [\48-\57\95\97-\122]

  Complement = ~[\n\r]
             = [\0-\255] / [\10\13]
             = [\0-\9\11-\12\14-\255]
```

Literals are Sequences of Characters

```
lexical syntax // literals

  Literal         = "then" // case sensitive sequence of characters

  CaseInsensitive = 'then' // case insensitive sequence of characters
```



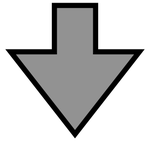
```
syntax

  "then" = [t] [h] [e] [n]
  'then' = [Tt] [Hh] [Ee] [Nn]
```



```
syntax

  "then" = [\116] [\104] [\101] [\110]
  'then' = [\84\116] [\72\104] [\69\101] [\78\110]
```

Identifiers

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
```

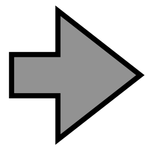


```
Id = a
Id = B
Id = cD
Id = xyz10
Id = internal_
Id = CamelCase
Id = lower_case
Id = ...
```

Lexical Ambiguity: Longest Match

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
context-free syntax
  Exp.Var  = Id
  Exp.Call = Exp Exp {left} // curried function call
```

ab

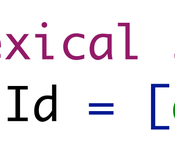


```
Mod(
  amb(
    [Var("ab"),
     Call(Var("a"), Var("b"))]
  )
)
```

Lexical Ambiguity: Longest Match

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
context-free syntax
  Exp.Var  = Id
  Exp.Call = Exp Exp {left} // curried function call
```

abc

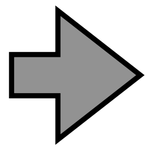


```
Mod(
  amb(
    [ amb(
        [ Var("abc")
        , Call(
            amb(
              [Var("ab"), Call(Var("a"), Var("b"))]
            )
          , Var("c")
          )
        ]
      )
    , Call(Var("a"), Var("bc"))
    ]
  )
)
```

Lexical Restriction => Longest Match

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
lexical restrictions
  Id -/- [a-zA-Z0-9\_] // longest match for identifiers
context-free syntax
  Exp.Var  = Id
  Exp.Call = Exp Exp {left} // curried function call
```

```
abc def ghi
```



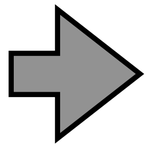
```
Call(Call(Var("abc"), Var("def")), Var("ghi"))
```

Lexical restriction: phrase cannot be followed by character in character class

Lexical Ambiguity: Keywords overlap with Identifiers

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
lexical restrictions
  Id -/- [a-zA-Z0-9\_] // longest match for identifiers
context-free syntax
  Exp.Var    = Id
  Exp.Call   = Exp Exp {left}
  Exp.IfThen = "if" Exp "then" Exp
```

```
if def then ghi
```

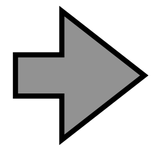


```
amb(
  [ Mod(
      Call(
        Call(Call(Var("if"), Var("def")), Var("then"))
      , Var("ghi")
      )
    )
  , Mod(IfThen(Var("def"), Var("ghi")))
  ]
)
```

Lexical Ambiguity: Keywords overlap with Identifiers

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
lexical restrictions
  Id -/- [a-zA-Z0-9\_] // longest match for identifiers
context-free syntax
  Exp.Var    = Id
  Exp.Call   = Exp Exp {left}
  Exp.IfThen = "if" Exp "then" Exp
```

```
ifdef then ghi
```

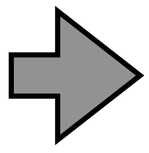


```
amb(
   [ Mod(
       Call(Call(Var("ifdef"), Var("then")), Var("ghi"))
     )
   , Mod(IfThen(Var("def"), Var("ghi")))
   ]
)
```

Reject Productions => Reserved Words

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
  Id = “if" {reject}
  Id = "then" {reject}
lexical restrictions
  Id -/- [a-zA-Z0-9\_] // longest match for identifiers
  "if" "then" -/- [a-zA-Z0-9\_]
context-free syntax
  Exp.Var    = Id
  Exp.Call   = Exp Exp {left}
  Exp.IfThen = "if" Exp "then" Exp
```

```
if def then ghi
```

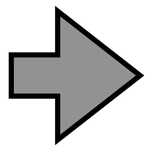


```
IfThen(Var("def"), Var("ghi"))
```

Reject Productions => Reserved Words

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
  Id = “if" {reject}
  Id = "then" {reject}
lexical restrictions
  Id -/- [a-zA-Z0-9\_] // longest match for identifiers
  "if" "then" -/- [a-zA-Z0-9\_]
context-free syntax
  Exp.Var    = Id
  Exp.Call   = Exp Exp {left}
  Exp.IfThen = "if" Exp "then" Exp
```

```
ifdef then ghi
```



```
parse error
```

Real Numbers

```
lexical syntax

  RealConst.RealConstNoExp = IntConst "." IntConst
  RealConst.RealConst = IntConst "." IntConst "e" Sign IntConst
  Sign = "+"
  Sign = "-"
```

Tiger Lexical Syntax

TU Delft

Tiger Lexical Syntax: Identifiers

```
module Identifiers

lexical syntax

  Id = [a-zA-Z] [a-zA-Z0-9\_]*

lexical restrictions

  Id -/- [a-zA-Z0-9\_]
```

Tiger Lexical Syntax: Number Literals

```
module Numbers

lexical syntax

  IntConst = [0-9]+

lexical syntax

  RealConst.RealConstNoExp = IntConst "." IntConst
  RealConst.RealConst = IntConst "." IntConst "e" Sign IntConst
  Sign = "+"
  Sign = "-"

context-free syntax

  Exp.Int = IntConst
```

Tiger Lexical Syntax: String Literals

```
module Strings

sorts StrConst

lexical syntax

  StrConst = "\"" StrChar* "\""
  StrChar  = ~[\\\"\n]
  StrChar  = [\\] [n]
  StrChar  = [\\] [t]
  StrChar  = [\\] [\^] [A-Z]
  StrChar  = [\\] [0-9] [0-9] [0-9]
  StrChar  = [\\] [\"]
  StrChar  = [\\] [\\]
  StrChar  = [\\] [\ \t\n]+ [\\]

context-free syntax // records

  Exp.String = StrConst
```

Tiger Lexical Syntax: Whitespace

```
module Whitespace

lexical syntax

  LAYOUT = [\ \t\n\r]

context-free restrictions

  // Ensure greedy matching for comments

  LAYOUT? -/- [\ \t\n\r]
  LAYOUT? -/- [\/].[\/]
  LAYOUT? -/- [\/].[\*]
```

Implicit composition of layout

```
syntax

  LAYOUT-CF = LAYOUT-LEX
  LAYOUT-CF = LAYOUT-CF LAYOUT-CF {left}
```

Tiger Lexical Syntax: Comment

```
lexical syntax
  CommentChar    = [\*]
  LAYOUT         = "/*" InsideComment* "*/"
  InsideComment  = ~[\*]
  InsideComment  = CommentChar
lexical restrictions
  CommentChar   -/- [\/]
context-free restrictions
  LAYOUT? -/- [\/].[\*]
```

```
lexical syntax
  LAYOUT             = SingleLineComment
  SingleLineComment  = "//" ~[\n\r]* NewLineEOF
  NewLineEOF         = [\n\r]
  NewLineEOF         = EOF
  EOF                =
lexical restrictions
  EOF     -/- ~[]
context-free restrictions
  LAYOUT? -/- [\/].[\/]
```

Desugaring Lexical Syntax

TU Delft

Explication of Lexical Syntax

Core language

- context-free grammar productions
- with constructors
- only character classes as terminals
- explicit definition of layout

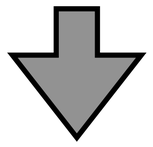
Desugaring

- express lexical syntax in terms of character classes
- explicate layout between context-free syntax symbols
- separate lexical and context-free syntax non-terminals

Explication of Layout by Transformation

```
context-free syntax

  Exp.Int    = IntConst
  Exp.Uminus = "-" Exp
  Exp.Times  = Exp "*" Exp {left}
  Exp.Divide = Exp "/" Exp {left}
  Exp.Plus   = Exp "+" Exp {left}
```

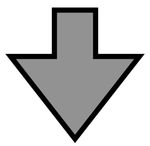
Symbols in context-free syntax are implicitly separated by optional layout



```
syntax

  Exp-CF.Int    = IntConst-CF
  Exp-CF.Uminus = "-" LAYOUT?-CF Exp-CF
  Exp-CF.Times  = Exp-CF LAYOUT?-CF "*" LAYOUT?-CF Exp-CF {left}
  Exp-CF.Divide = Exp-CF LAYOUT?-CF "/" LAYOUT?-CF Exp-CF {left}
  Exp-CF.Plus   = Exp-CF LAYOUT?-CF "+" LAYOUT?-CF Exp-CF {left}
```

Separation of Lexical and Context-free Syntax

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
  Id = “if" {reject}
  Id = "then" {reject}
context-free syntax
  Exp.Var = Id
```



```
syntax

  Id-LEX = [\65-\90\97-\122] [\48-\57\65-\90\95\97-\122]*-LEX
  Id-LEX = "if" {reject}
  Id-LEX = "then" {reject}
  Id-CF  = Id-LEX

  Exp-CF.Var = Id-CF
```

```
lexical syntax
  Id = [a-zA-Z] [a-zA-Z0-9\_]*
  Id = “if" {reject}
  Id = "then" {reject}
lexical restrictions
  Id -/- [a-zA-Z0-9\_]
  "if" "then" -/- [a-zA-Z0-9\_]
context-free syntax
  Exp.Var    = Id
  Exp.Call   = Exp Exp {left}
  Exp.IfThen = "if" Exp "then" Exp
```

character classes as only terminals

separate lexical and context-free syntax

separate context-free symbols by optional layout

```
syntax

  "if"   = [\105] [\102]
  "then" = [\116] [\104] [\101] [\110]

  [\48-\57\65-\90\95\97-\122]+-LEX = [\48-\57\65-\90\95\97-\122]
  [\48-\57\65-\90\95\97-\122]+-LEX = [\48-\57\65-\90\95\97-\122]+-LEX [\48-\57\65-\90\95\97-\122]
  [\48-\57\65-\90\95\97-\122]*-LEX =
  [\48-\57\65-\90\95\97-\122]*-LEX = [\48-\57\65-\90\95\97-\122]+-LEX

  Id-LEX = [\65-\90\97-\122] [\48-\57\65-\90\95\97-\122]*-LEX
  Id-LEX = "if" {reject}
  Id-LEX = "then" {reject}
  Id-CF  = Id-LEX

  Exp-CF.Var    = Id-CF
  Exp-CF.Call   = Exp-CF LAYOUT?-CF Exp-CF {left}
  Exp-CF.IfThen = "if" LAYOUT?-CF Exp-CF LAYOUT?-CF "then" LAYOUT?-CF Exp-CF

  LAYOUT-CF  = LAYOUT-CF LAYOUT-CF {left}
  LAYOUT?-CF = LAYOUT-CF
  LAYOUT?-CF =

restrictions

  Id-LEX -/- [\48-\57\65-\90\95\97-\122]
  "if"   -/- [\48-\57\65-\90\95\97-\122]
  "then" -/- [\48-\57\65-\90\95\97-\122]

priorities

  Exp-CF.Call left Exp-CF.Call,
  LAYOUT-CF = LAYOUT-CF LAYOUT-CF  left LAYOUT-CF = LAYOUT-CF LAYOUT-CF
```

Exam Question

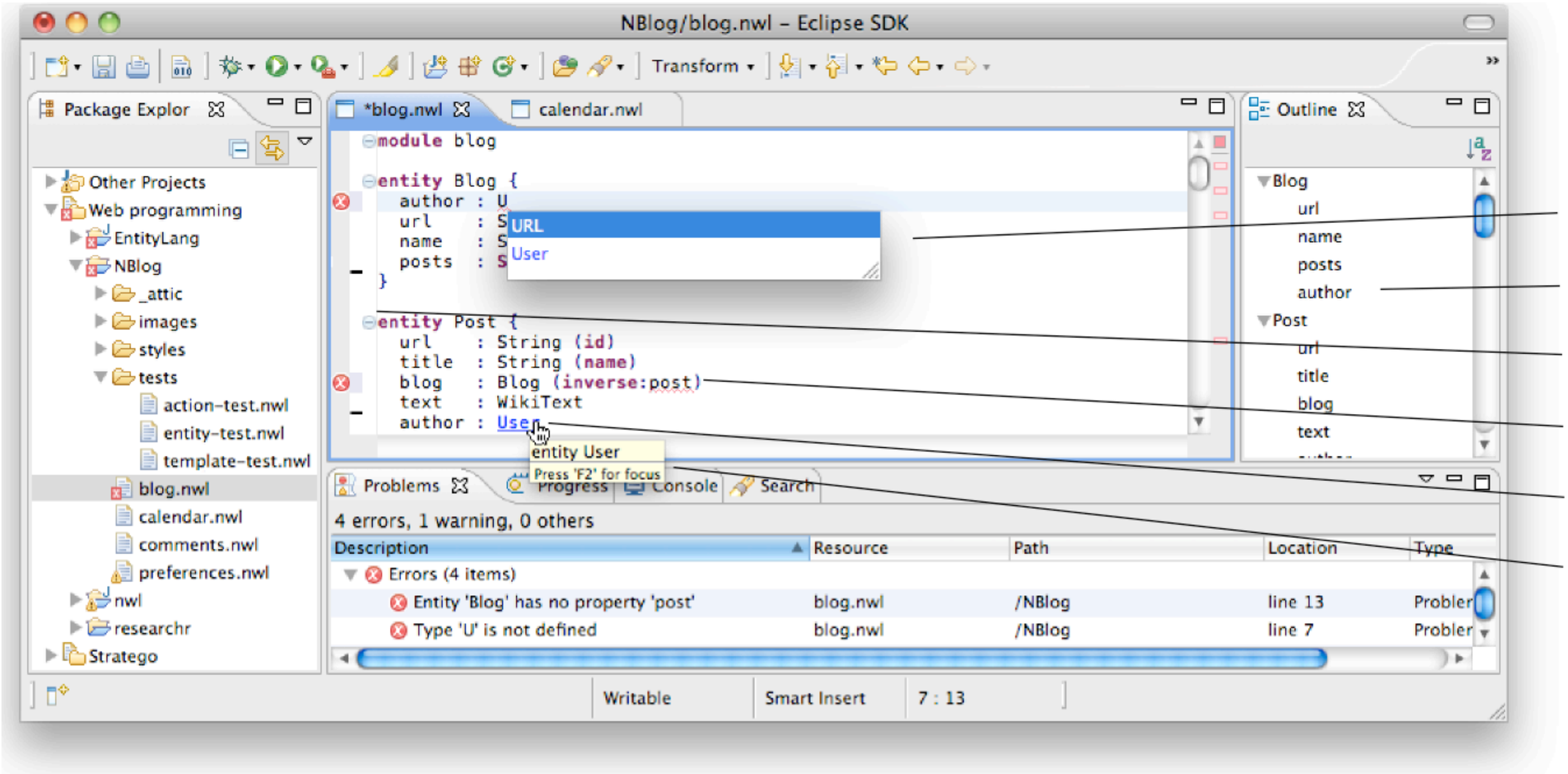
Give an example syntax definition that demonstrates the need for separating lexical and context-free syntax

Explain what goes wrong in the absence of that separation

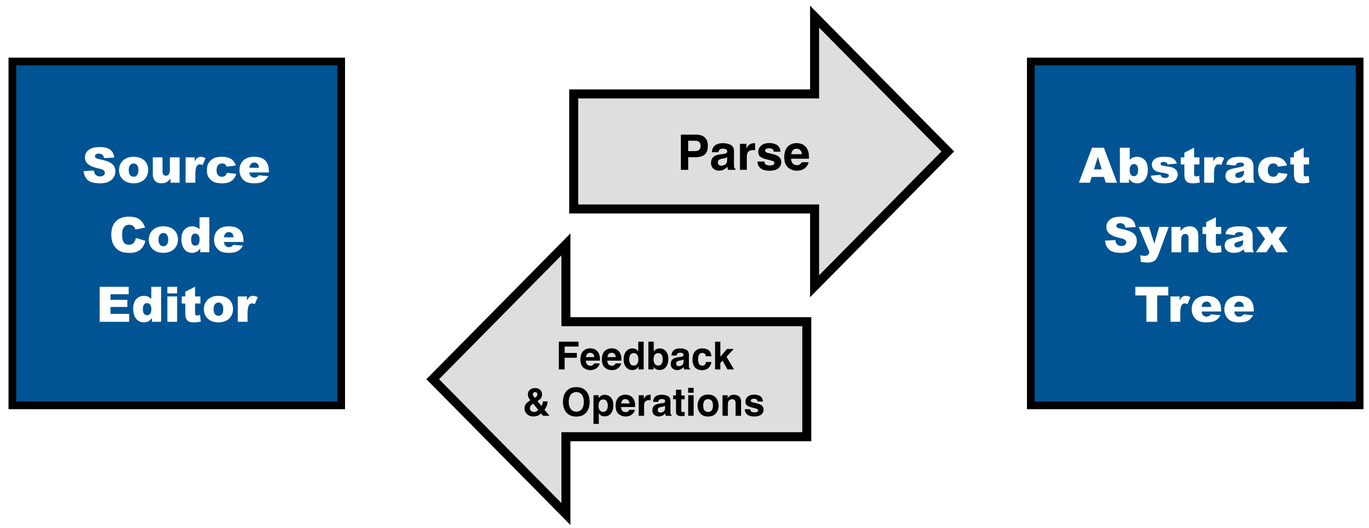
Syntactic (Editor) Services

What can we do with a syntax definition?

Editor Services



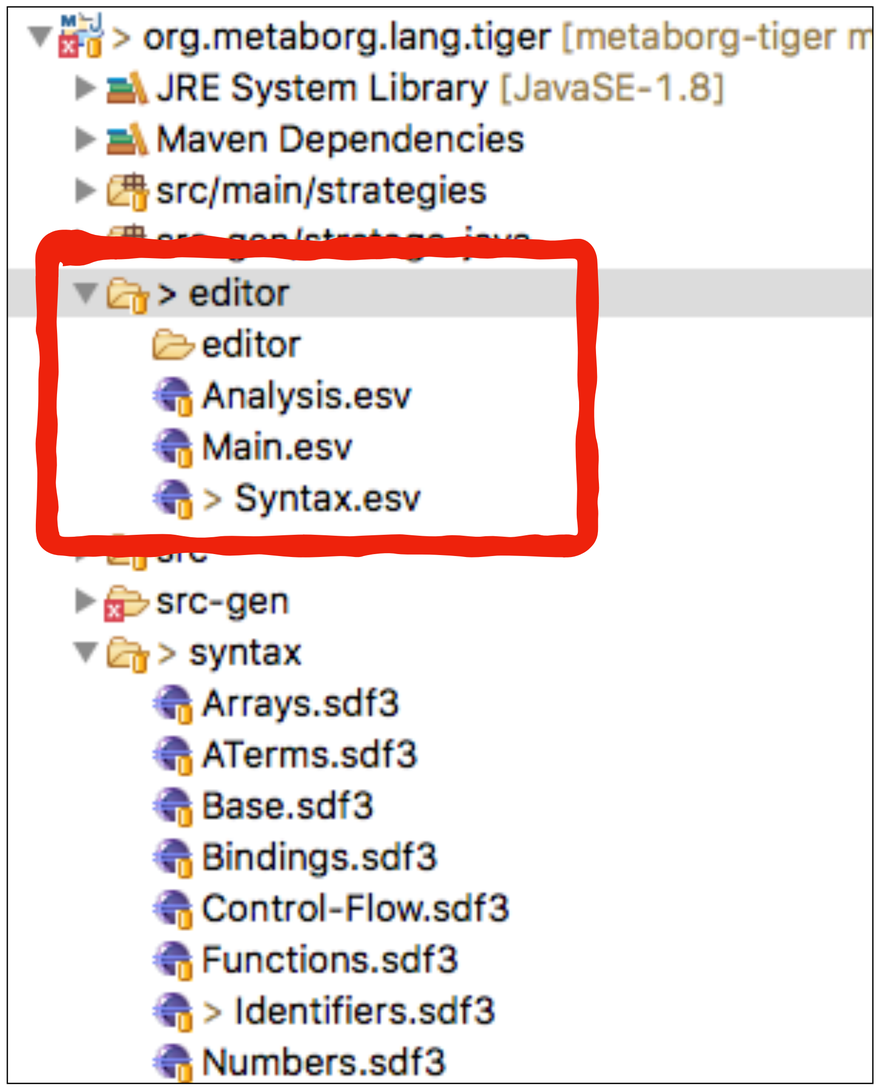
Editor Services



Feedback
- syntax coloring
- syntax checking
- outline view

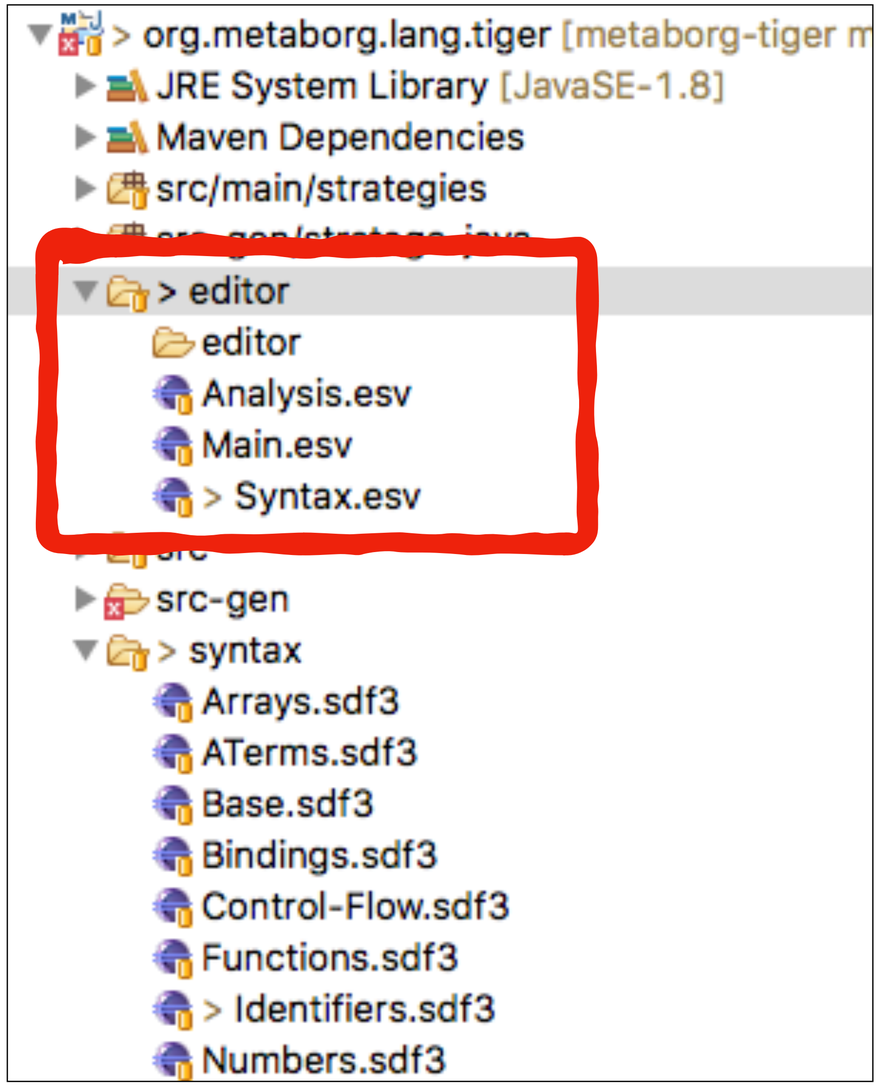
Operations
- syntactic completion
- formatting
- abstract syntax tree

Language Project Configuration (ESV)




```
module Main

imports

  Syntax
  Analysis

language

  extensions : tig

  //provider : target/metaborg/stratego.ctree
  provider : target/metaborg/stratego.jar
  provider : target/metaborg/stratego-javastrat.jar

menus

  menu: "Transform" (openeditor) (realtime)

    action: "Desugar"         = editor-desugar (source)
    action: "Desugar AST"     = editor-desugar-ast (source)
```

Configuration of Syntactic Services (ESV)




```
module Syntax

imports

  libspoofax/color/default
  completion/colorer/Tiger-cc-esv

language

  table          : target/metaborg/sdf-new.tbl
  //table        : target/metaborg/sdf.tbl
  start symbols : Module

  line comment  : "//"
  block comment : "/*" * "*/"
  fences        : [ ] ( ) { }

menus

  menu: "Syntax" (openeditor)

    action: "Format"           = editor-format (source)
    action: "Show parsed AST" = debug-show-aterm (source)

views

  outline view: editor-outline (source)
    expand to level: 3
```

Syntax Coloring

Generated Syntax Coloring

```
module libspoofax/color/default

imports

  libspoofax/color/colors

colorer // Default, token-based highlighting

  keyword    : 127 0 85 bold
  identifier : default
  string     : blue
  number     : darkgreen
  var        : 139 69 19 italic
  operator   : 0 0 128
  layout     : 63 127 95 italic
```

```
// compute the n-th fibonacci number
let function fib(n: int): int =
      if n <= 1 then 1
      else fib(n - 1) + fib(n - 2)
 in fib(10)
end
```

Customized Syntax Coloring

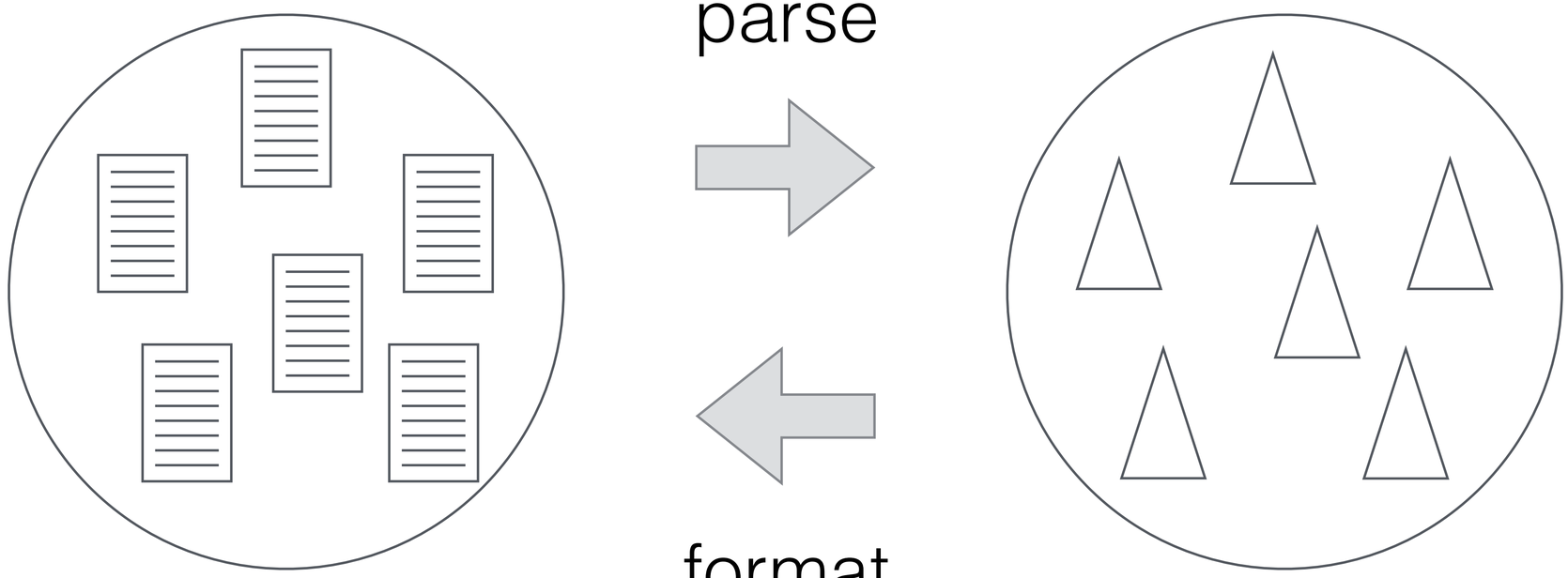
```
module Tiger-Colorer

colorer

  red   = 255 0 0
  green = 0 255 0
  blue  = 0 0 255
  TUDlavender = 123 160 201

colorer token-based highlighting

   keyword  : red
   Id       : TUDlavender
   StrConst : darkgreen
   TypeId   : blue
   layout   : green
```

```
// compute the n-th fibonacci number
let function fib(n: int): int =
      if n <= 1 then 1
      else fib(n - 1) + fib(n - 2)
 in fib(10)
end
```

From Unparsers to Pretty-Printers with Templates

TU Delft

Language = Sentences and Trees

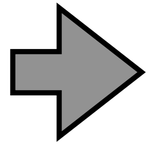


Unparsing: From Abstract Syntax Term to Text

```
context-free syntax
  Exp.Int   = IntConst
  Exp.Times = [[Exp] * [Exp]] {left}
  Exp.Plus  = [[Exp] + [Exp]] {left}
```

```
rules

  unparse :
    Int(x) -> x
  unparse :
    Plus(e1, e2) -> $[[<unparse> e1] + [<unparse> e2]]
  unparse :
    Times(e1, e2) -> $[[<unparse> e1] * [<unparse> e2]]
```

```
Mod(
  Plus(
    Int("1")
  , Times(
      Int("2")
    , Plus(Int("3"), Int("4"))
    )
  )
)
```



```
1 + 2 * 3 + 4
```

take priorities into account!

Pretty-Printing

From ASTs to text

- insert keywords
- insert layout: spaces, line breaks, indentation
- insert parentheses to preserve tree structure

Unparser

- derive transformation rules from context-free grammar
- keywords, literals defined in grammar productions
- parentheses determined by priority, associativity rules
- separate all symbols by a space => not pretty, or even readable

Pretty-printer

- introduce spaces, line breaks, and indentation to produce readable text
- doing that manually is tedious

Specifying Formatting Layout with Templates

```
context-free syntax

  Exp.Seq = <
    (
      <{Exp ";\n"}*>
    )
  >

  Exp.If = <
    if <Exp> then
      <Exp>
    else
      <Exp>
  >

  Exp.IfThen = <
    if <Exp> then
      <Exp>
  >

  Exp.While = <
    while <Exp> do
      <Exp>
  >
```

Inverse quotation
- template quotes literal text with <>
- anti-quotations insert non-terminals with <>

Layout directives
- whitespace (linebreaks, indentation, spaces) in template guides formatting
- is interpreted as LAYOUT? for parsing

Formatter generation
- generate rules for mapping AST to text (via box expressions)

Applications
- code generation; pretty-printing generated AST
- syntactic completions
- formatting

Templates for Tiger: Binary Expressions

```
context-free syntax

  Exp.Int      = IntConst

  Exp.Uminus   = [- [Exp]]
  Exp.Times    = [[Exp] * [Exp]]    {left}
  Exp.Divide   = [[Exp] / [Exp]]    {left}
  Exp.Plus     = [[Exp] + [Exp]]    {left}
  Exp.Minus    = [[Exp] - [Exp]]    {left}

  Exp.Eq       = [[Exp] = [Exp]]    {non-assoc}
  Exp.Neq      = [[Exp] <> [Exp]]   {non-assoc}
  Exp.Gt       = [[Exp] > [Exp]]    {non-assoc}
  Exp.Lt       = [[Exp] < [Exp]]    {non-assoc}
  Exp.Geq      = [[Exp] >= [Exp]]   {non-assoc}
  Exp.Leq      = [[Exp] <= [Exp]]   {non-assoc}

  Exp.And      = [[Exp] & [Exp]]    {left}
  Exp.Or       = [[Exp] | [Exp]]    {left}
```

Use [] quotes instead of <> to avoid clash with comparison operators

Templates for Tiger: Functions

Function declarations separated by newline

```
context-free syntax

  Dec.FunDecs = <<{FunDec "\n"}+>> {longest-match}

  FunDec.ProcDec = <
    function <Id>(<{FArg ", "}*>) =
      <Exp>
  >

  FunDec.FunDec = <
    function <Id>(<{FArg ", "}*>) : <Type> =
      <Exp>
  >

  FArg.FArg = <<Id> : <Type>>

  Exp.Call = <<Id>(<{Exp ", "}*>)>
```

Indent body of function

Space after comma!

No space after function name in call

Templates for Tiger: Bindings and Records

```
context-free syntax

  Exp.Let = <
    let
      <{Dec "\n"}*>
     in
      <{Exp ";\n"}*>
    end
  >
```

```
context-free syntax // records

  Type.RecordTy = <
    {
       <{Field ", \n"}*>
    }
  >

  Field.Field = <<Id> : <TypeId>>

  Exp.NilExp = <nil>

  Exp.Record = <<TypeId>{ <{InitField ", "}*> }>

  InitField.InitField = <<Id> = <Exp>>

  LValue.FieldVar = <<LValue>.<Id>>
```

Note spacing / layout in separators

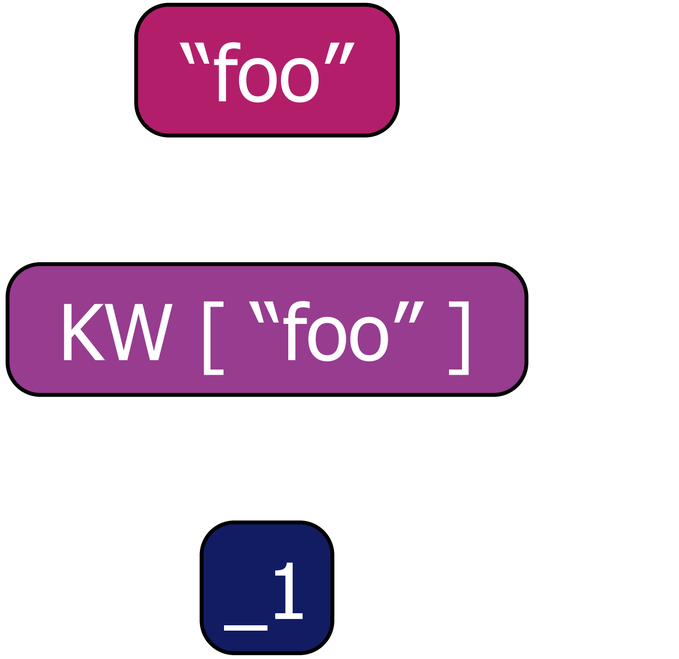
Generating Pretty-Print Rules from Template Productions

```
context-free syntax

  FunDec.FunDec = <
    function <Id>(<{FArg ", "}*>) : <Type> =
      <Exp>
  >
```

Separation of concerns:
- generated formatter transforms AST to Box
- Box formatter produces text

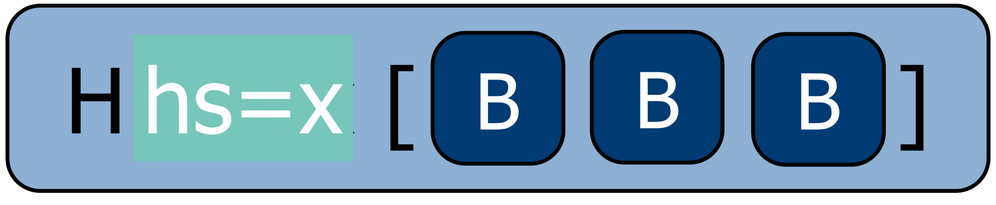
```
rules

  prettyprint-Tiger-FunDec :
    ProcDec(t1__, t2__, t3__) -> [ H(
                                     [SOpt(HS(), "0")]
                                   , [ S("function ")
                                     , t1__
                                     , S("(")
                                     , t2__'
                                     , S(") =")
                                     ]
                                   )
                                 , t3__'
                                 ]
    with t1__' := <pp-one-Z(prettyprint-Tiger-Id) <+ pp-one-Z(prettyprint-completion-aux)> t1__
    with t2__' := <pp-H-list(prettyprint-Tiger-FArg|", ")
                    <+ pp-one-Z(prettyprint-completion-aux)> t2__
    with t3__' := <pp-indent(|"2")> [ <pp-one-Z(prettyprint-Tiger-Exp) <+ pp-one-Z(prettyprint-completion-aux)> t3__
                                    ]
```

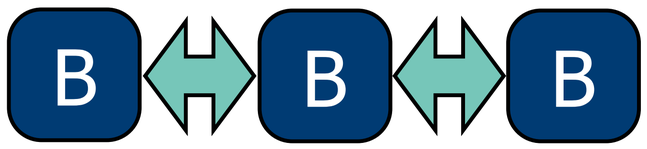
Boxes for Formatting



literal text, keywords, parameters

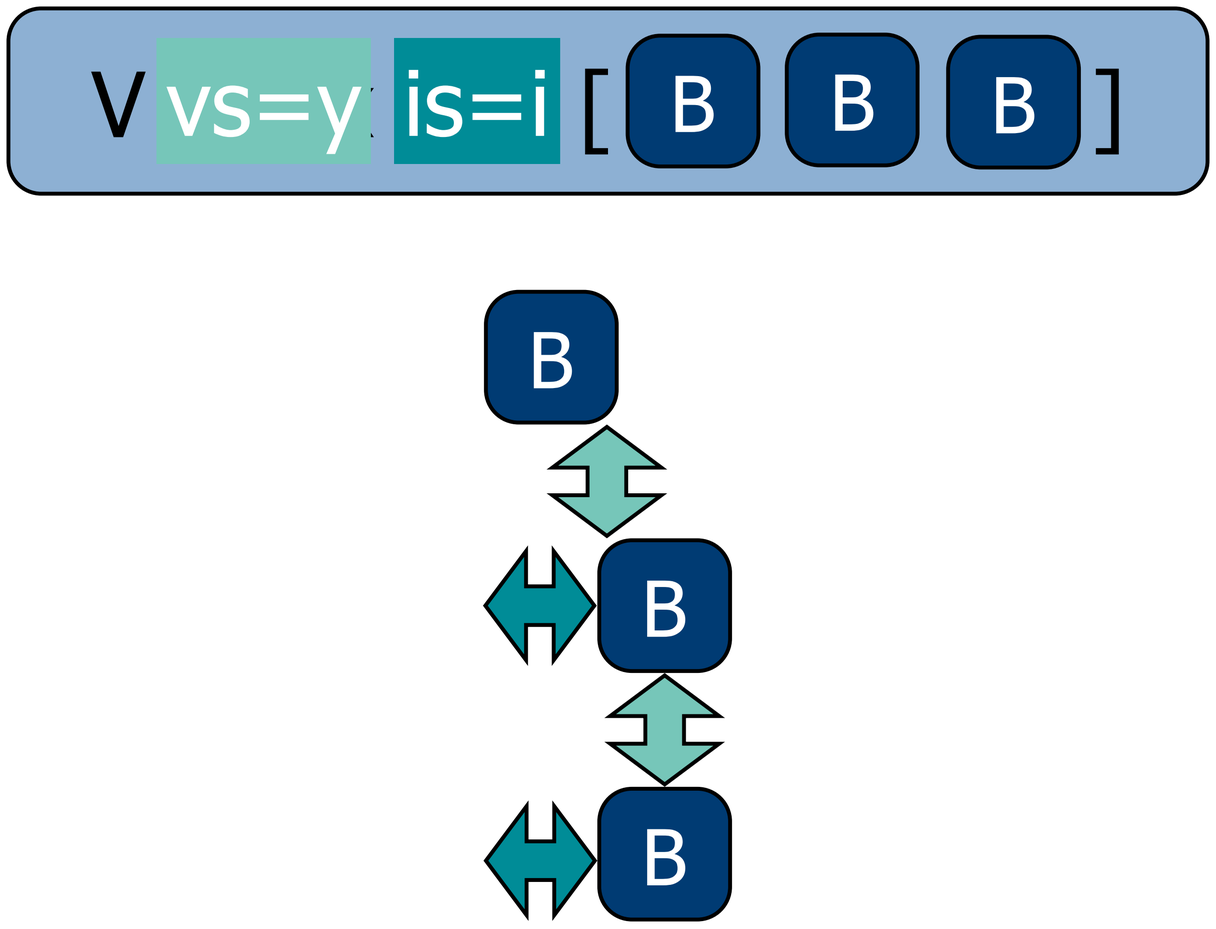
Horizontal Layout





hs: horizontal space between boxes

Vertical Layout



vs: vertical space between boxes; is: indentation space

Tiger Syntax Definition with Templates

Tiger Syntax: Composition

```
module Tiger

imports Whitespace
imports Comments
imports Types
imports Identifiers
imports Bindings
imports Variables
imports Functions
imports Numbers
imports Strings
imports Records
imports Arrays
imports Control-Flow

context-free start-symbols Module

context-free syntax

  Module.Mod = Exp

context-free priorities

  Exp.Or > Exp.Array > Exp.Assign ,

  {Exp.Uminus LValue.FieldVar LValue.Subscript}
  > {left : Exp.Times Exp.Divide}
```

Tiger Syntax: Identifiers and Strings

```
module Identifiers

lexical syntax

  Id = [a-zA-Z] [a-zA-Z0-9\_]*

lexical restrictions

  Id -/- [a-zA-Z0-9\_]

lexical syntax

  Id = "nil" {reject}
  Id = "let" {reject}
  Id = … {reject}
```

```
module Strings

sorts StrConst

lexical syntax

  StrConst = "\"" StrChar* "\""
  StrChar = ~[\\\"\n]
  StrChar = [\\] [n]
  StrChar = [\\] [t]
  StrChar = [\\] [\^] [A-Z]
  StrChar = [\\] [0-9] [0-9] [0-9]
  StrChar = [\\] [\"]
  StrChar = [\\] [\\]
  StrChar = [\\] [\ \t\n]+ [\\]

context-free syntax // records

  Exp.String = StrConst
```

Tiger Syntax: Whitespace

```
module Whitespace

lexical syntax

  LAYOUT = [\ \t\n\r]

context-free restrictions

  LAYOUT? -/- [\ \t\n\r]
```

Tiger Syntax: Comments

```
module Comments

lexical syntax // multiline comments
  CommentChar    = [\*]
  LAYOUT         = "/*" InsideComment* "*/"
  InsideComment  = ~[\*]
  InsideComment  = CommentChar

lexical restrictions
  CommentChar -/- [\/]

context-free restrictions
  LAYOUT? -/- [\/].[\/]

lexical syntax // single line comments
  LAYOUT      = "//" ~[\n\r]* NewLineEOF
  NewLineEOF  = [\n\r]
  NewLineEOF  = EOF
  EOF         =
  // end of file since it cannot be followed by any character
  // avoids the need for a newline to close a single line comment
  // at the last line of a file

lexical restrictions
  EOF -/- ~[]

context-free restrictions
  LAYOUT? -/- [\/].[\*]
```

Tiger Syntax: Numbers

```
module Numbers

lexical syntax

  IntConst = [0-9]+

context-free syntax

  Exp.Int     = IntConst

  Exp.Uminus  = [- [Exp]]
  Exp.Times   = [[Exp] * [Exp]]   {left}
  Exp.Divide  = [[Exp] / [Exp]]   {left}
  Exp.Plus    = [[Exp] + [Exp]]   {left}
  Exp.Minus   = [[Exp] - [Exp]]   {left}

  Exp.Eq      = [[Exp] = [Exp]]   {non-assoc}
  Exp.Neq     = [[Exp] <> [Exp]]  {non-assoc}
  Exp.Gt      = [[Exp] > [Exp]]   {non-assoc}
  Exp.Lt      = [[Exp] < [Exp]]   {non-assoc}
  Exp.Geq     = [[Exp] >= [Exp]]  {non-assoc}
  Exp.Leq     = [[Exp] <= [Exp]]  {non-assoc}

  Exp.And     = [[Exp] & [Exp]]   {left}
  Exp.Or      = [[Exp] | [Exp]]   {left}
```

```
context-free priorities

  {Exp.Uminus}
  > {left :
    Exp.Times
    Exp.Divide}
  > {left :
    Exp.Plus
    Exp.Minus}
  > {non-assoc :
    Exp.Eq
    Exp.Neq
    Exp.Gt
    Exp.Lt
    Exp.Geq
    Exp.Leq}
  > Exp.And
  > Exp.Or
```

Tiger Syntax: Variables and Functions

```
module Bindings

imports Control-Flow
imports Identifiers
imports Types
imports Functions
imports Variables

sorts Declarations

context-free syntax

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

  Declarations.Declarations = <
    declarations <{Dec "\n"}*>
  >
```

```
module Variables

imports Identifiers
imports Types

sorts Var

context-free syntax

  Dec.VarDec = <var <Id> : <Type> := <Exp>>


  Dec.VarDecNoType = <var <Id> := <Exp>>
  Var.Var = Id

  LValue = Var

  Exp = LValue

  Exp.Assign = <<LValue> := <Exp>>
```

```
module Functions

imports Identifiers
imports Types

context-free syntax

  Dec.FunDecs = <<{FunDec "\n"}+>> {longest-match}

  FunDec.ProcDec = <
    function <Id>(<{FArg ", "}*>) =
      <Exp>
  >

  FunDec.FunDec = <
    function <Id>(<{FArg ", "}*>) : <Type> =
      <Exp>
  >

  FArg.FArg = <<Id> : <Type>>

  Exp.Call = <<Id>(<{Exp ", "}*>)>
```

Tiger Syntax: Records, Arrays, Types

```
module Records
imports Base
imports Identifiers
imports Types
context-free syntax // records

  Type.RecordTy = <
    {
       <{Field ", \n"}*>
    }
  >

  Field.Field = <<Id> : <TypeId>>

  Exp.NilExp = <nil>

  Exp.Record = <<TypeId>{ <{InitField ", "}*> }>

  InitField.InitField = <<Id> = <Exp>>

  LValue.FieldVar = <<LValue>.<Id>>
```

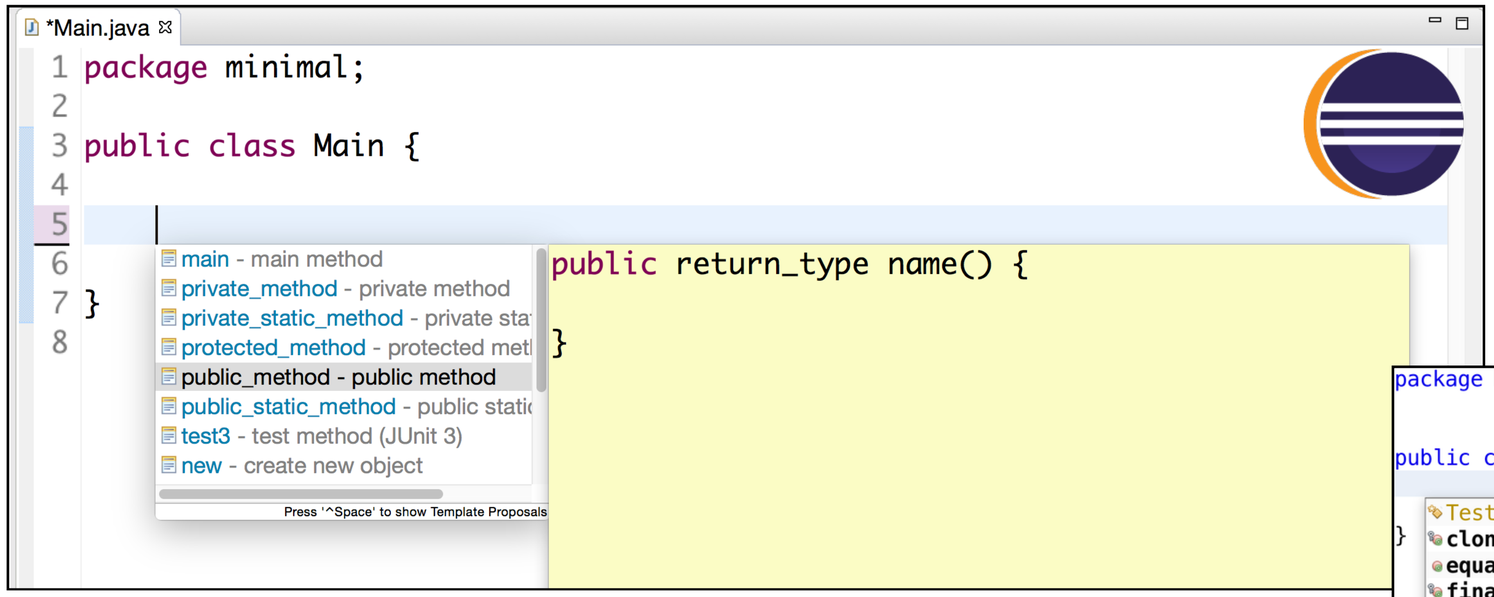
```
module Arrays
imports Types
context-free syntax // arrays

  Type.ArrayTy = <array of <TypeId>>

  Exp.Array = <<TypeId>[<Exp>] of <Exp>>

  LValue.Subscript = <<LValue>[<Index>]>

  Index = Exp
```

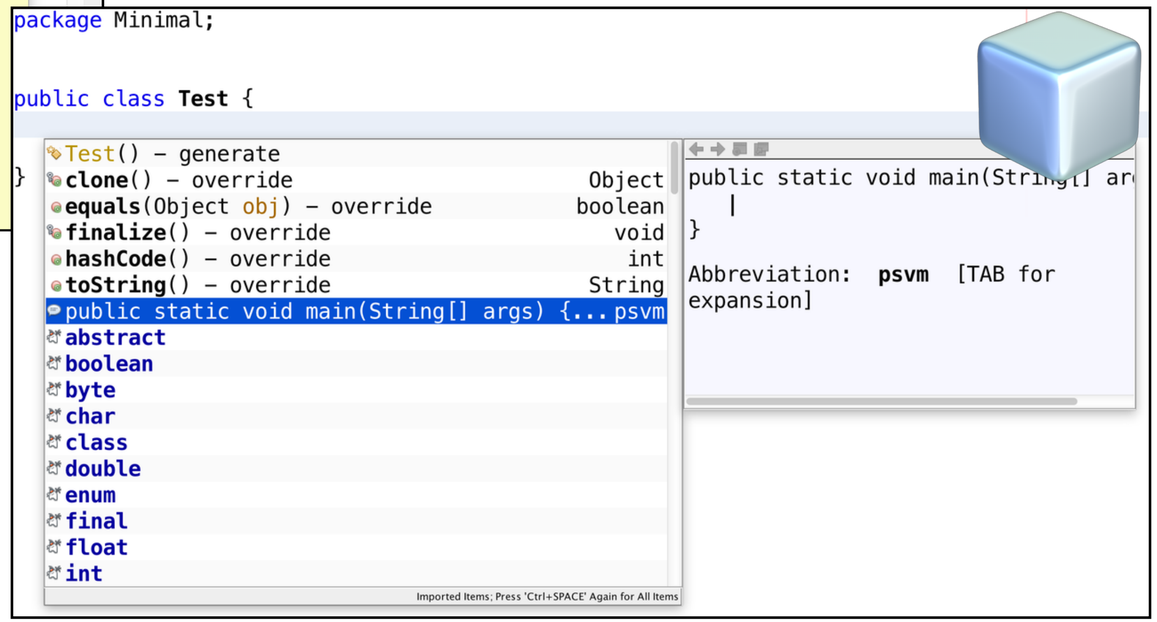
```
module Types

imports Identifiers
imports Bindings

sorts Type

context-free syntax // type declarations

  Dec.TypeDecs = <<{TypeDec "\n"}+>> {longest-match}

  TypeDec.TypeDec = <type <Id> = <Type>>

context-free syntax // type expressions

  Type = TypeId
  TypeId.Tid  = Id

sorts Ty
context-free syntax // semantic types

  Ty.INT       = <INT>
  Ty.STRING    = <STRING>
  Ty.NIL       = <NIL>
  Ty.UNIT      = <UNIT>
  Ty.NAME      = <NAME <Id>>
  Ty.RECORD    = <RECORD <Id>>
  Ty.ARRAY     = <ARRAY <Ty> <Id>>
  Ty.FUN       = <FUN ( <{Ty ","}*> ) <Ty>>
```

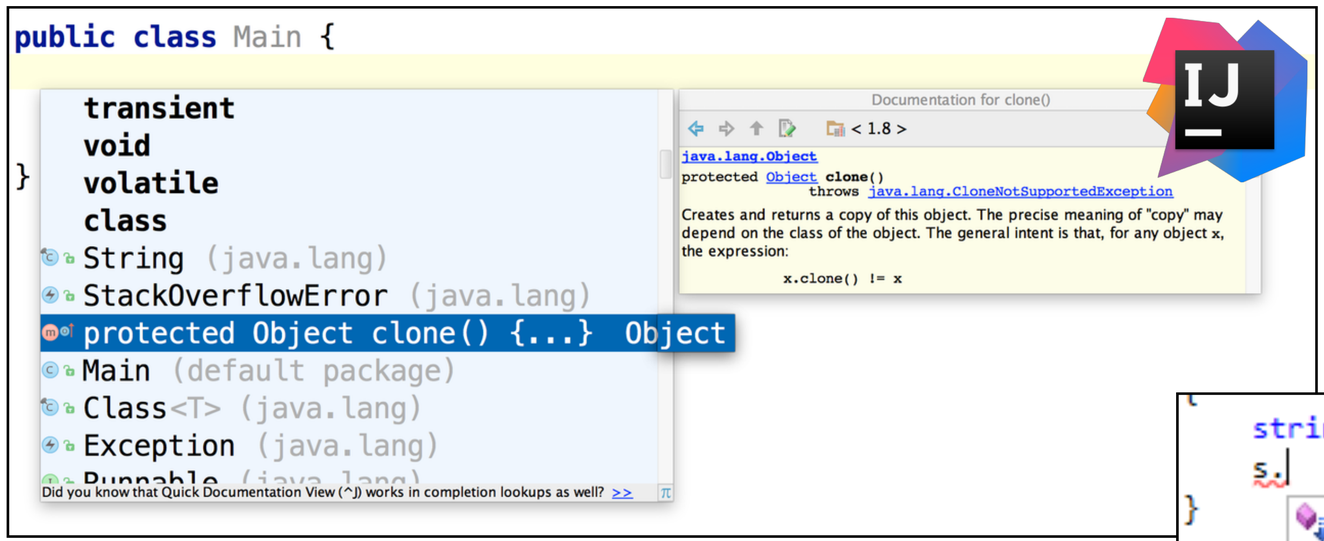
Syntactic Completion

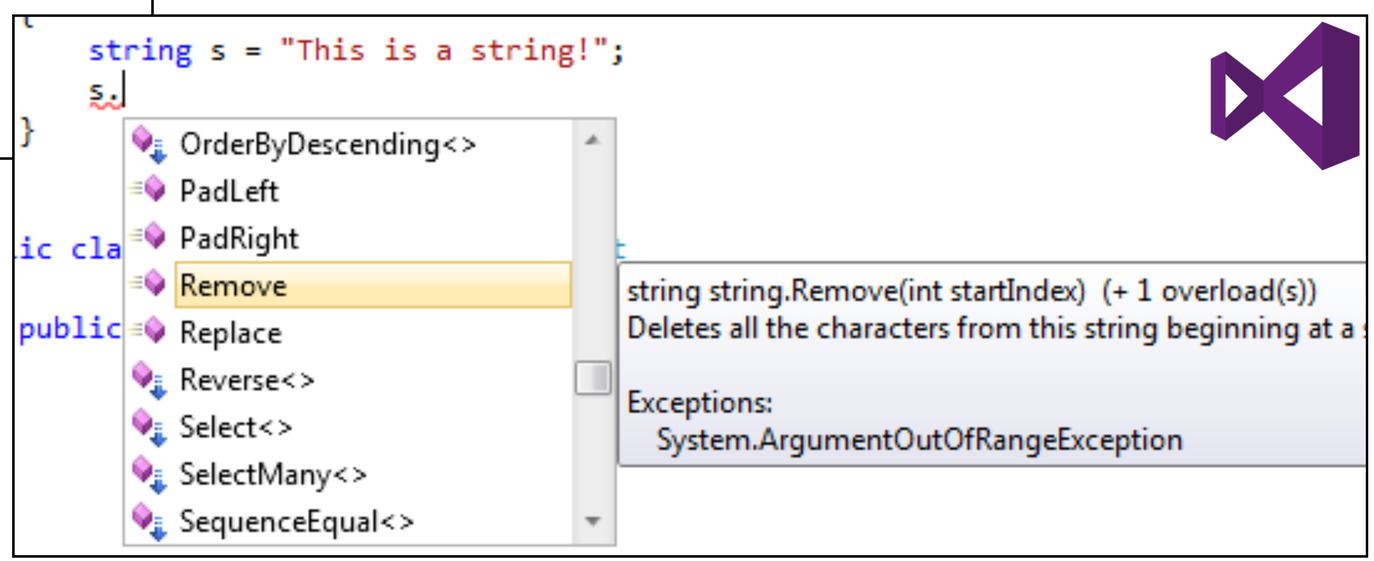
Amorim, Erdweg, Wachsmuth, Visser
Principled syntactic code completion using placeholders.
In ACM SIGPLAN International Conference on Software Language Engineering [SLE 2016].

TU Delft









*S. Amann, S. Proksch, S. Nadi, and M. Mezini. A study of visual studio usage in practice. In SANER, 2016.

```
public class Main {

}
```

```
private_method - private m
private_static_method - pri
protected_method - protec
public_method - public me
public_static_method - pub
main - main method
```

Press '^Space' to show SWT Template Proposals

```
public static void main(String[] args) {

}
```

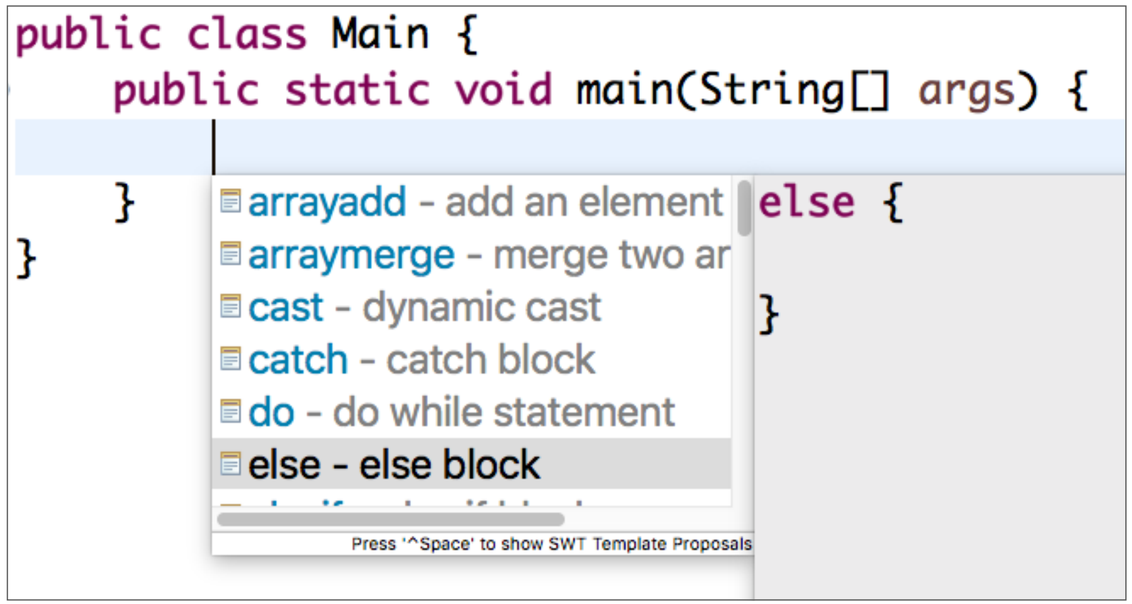
Syntactic Completion

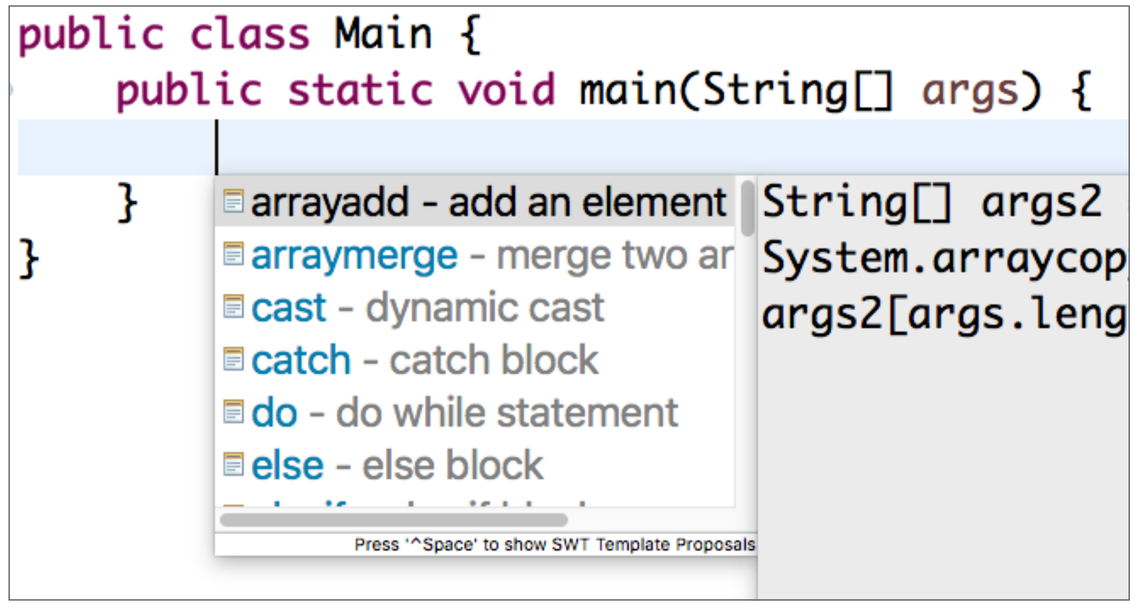
```
public class Fibonacci {
    static long fibo(int n) {
        return (n < 2) ? n : fibo(n - 1) + fibo(n - 2);
    }

    public static void main(String[] args){
        for (int i = 0; i < 30; i++){
            System.out.print("(" + i + "):" + Fibonacci. + "\t");
        }
    }
}
```

```
fibo(int n) : long - Fibonacci
class : Class<slepresentatior
main(String[] args) : void - Fi
```

Semantic Completion

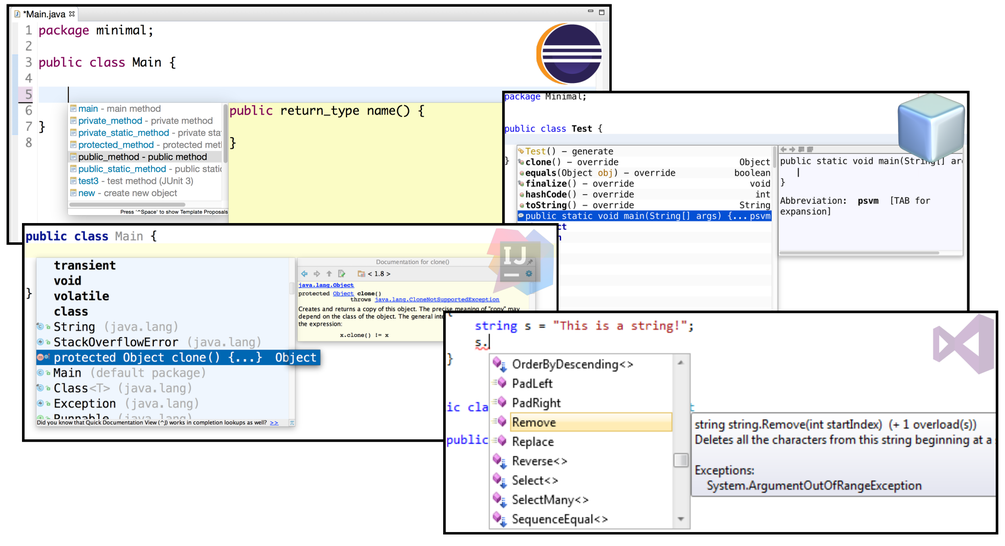
Problems





Unsound: propose invalid constructs

Incomplete: not all programs reachable



Ad-hoc: re-implement for each language / IDE

Goal

Sound and ***Complete***
Syntactic Code Completion
from ***Syntax Definition***

Idea 1

Explicit Placeholders

Incomplete programs

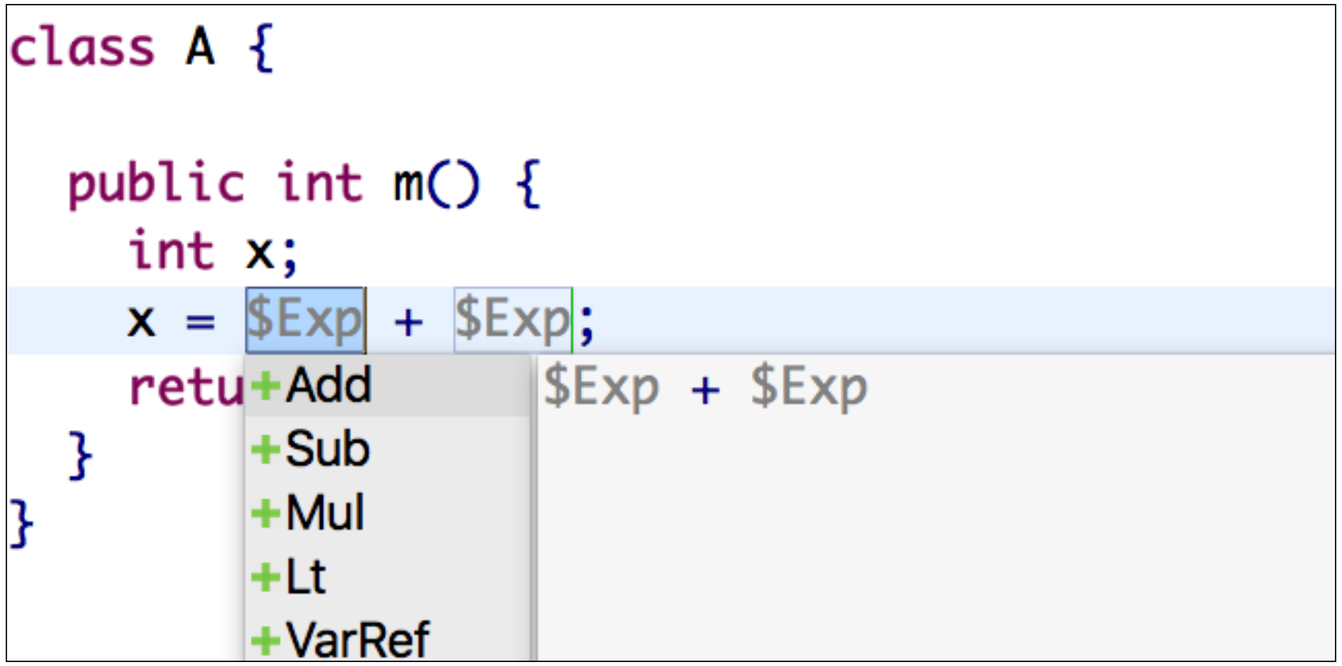
```
class A {

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

Incomplete programs

```
class A {

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

Make incompleteness valid using placeholders!

```
class A {

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


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

```
class A {

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

```
+Add    ($Exp + $Exp)
+Sub
+Mul
+Lt
+VarRef
```

```
class A {

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

1

```
class A {

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

+Add
+Sub
+Mul
+Lt
+VarRef

$Exp + $Exp

2

```
class A {

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

+Add
+Sub
+Mul
+Lt
+VarRef

$Exp + $Exp

3

```
class A {

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

+Add
+Sub
+Mul
+Lt
+VarRef

($Exp + $Exp)

4

```
class A {

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

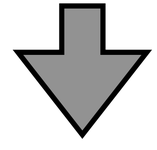
Deriving Syntactic Completion from Syntax Definition

```
context-free syntax  // regular production

  Statement.If = "if" "(" Exp ")" Statement "else" Statement
```

Placeholders as Language Constructs

```
context-free syntax  // regular production

  Statement.If = "if" "(" Exp ")" Statement "else" Statement
```

```
context-free syntax // placeholder rule

  Statement.Statement-Plhdr = “$Statement"
```



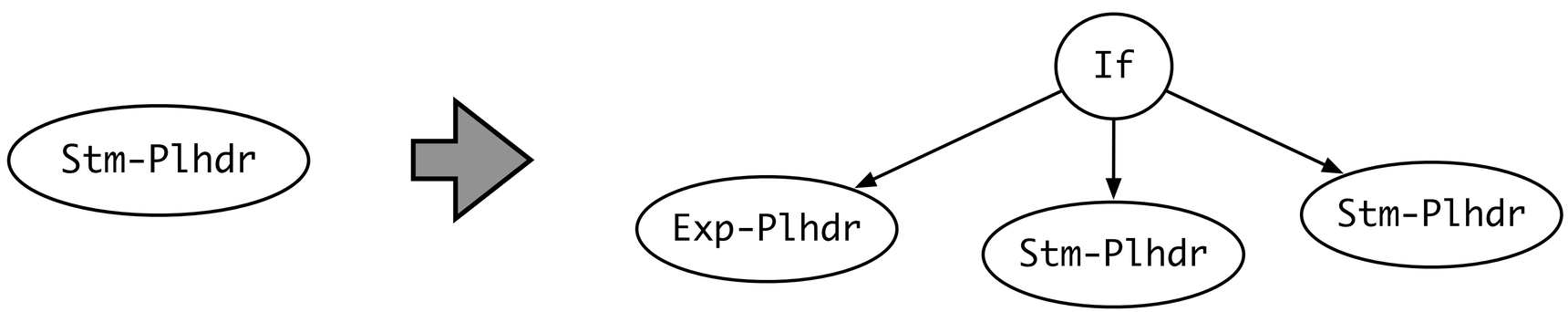
```
class A {

  public int m() {
    int x;
    if(x < 0) x = 0;
    else $Statement
    return x;
  }
}
```

Calculate Placeholder Expansions

```
context-free syntax  // regular production

  Statement.If = "if" "(" Exp ")" Statement "else" Statement
```

```
context-free syntax // placeholder rule

  Statement.Statement-Plhdr = “$Statement"
```

```
rules

  rewrite-placeholder:
    Statement-Plhdr() -> If(Exp-Plhdr(), Statement-Plhdr(),
                            Statement-Plhdr())
```



Calculate Placeholder Expansions

```
context-free syntax  // regular production

  Statement.If = "if" "(" Exp ")" Statement "else" Statement
```

```
context-free syntax // placeholder rule

  Statement.Statement-Plhdr = “$Statement"
```

```
rules

  rewrite-placeholder:
    Statement-Plhdr() -> If(Exp-Plhdr(), Statement-Plhdr(),
                            Statement-Plhdr())
```

```
class A {

  public int m() {
    int x;
    if(x < 0) x = 0;
    else $Statement
    return x;
  }
}
```

+VarDecl
+Assign
+Block
+If
+While

```
if($Exp) $Statement
else $Statement
```

Correct programs

Complete programs

Incomplete programs

Expand placeholder

Expand/overwrite placeholders

Complete programs

```
class A {

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

How to expand a complete program?

```
class A {

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

How to expand a complete program?

```
class A $ParentDecl {

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

+Parent

extends $ID

```
class A extends $ID {

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

Insert a placeholder?

```
class A {

  public int m() {
    $Statement
    int
    x =
    ret
  }
}
```

+VarDecl

+Assign

+Block

+If

+While

$Type $ID;

```
class A {

  public int m() {
    $Type $ID;

  }
}
```

+Int

+Bool

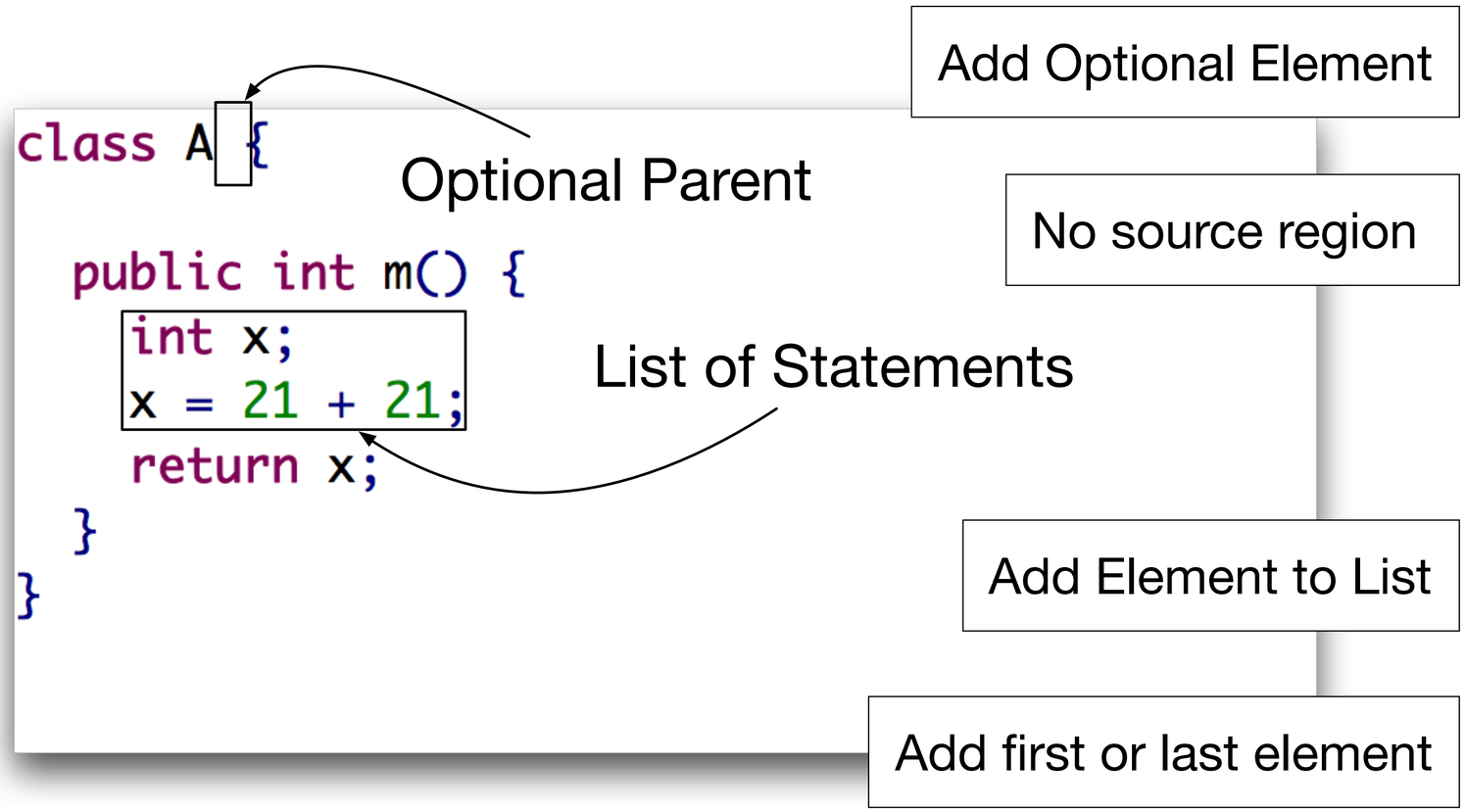
+ClassType

int

Idea 2

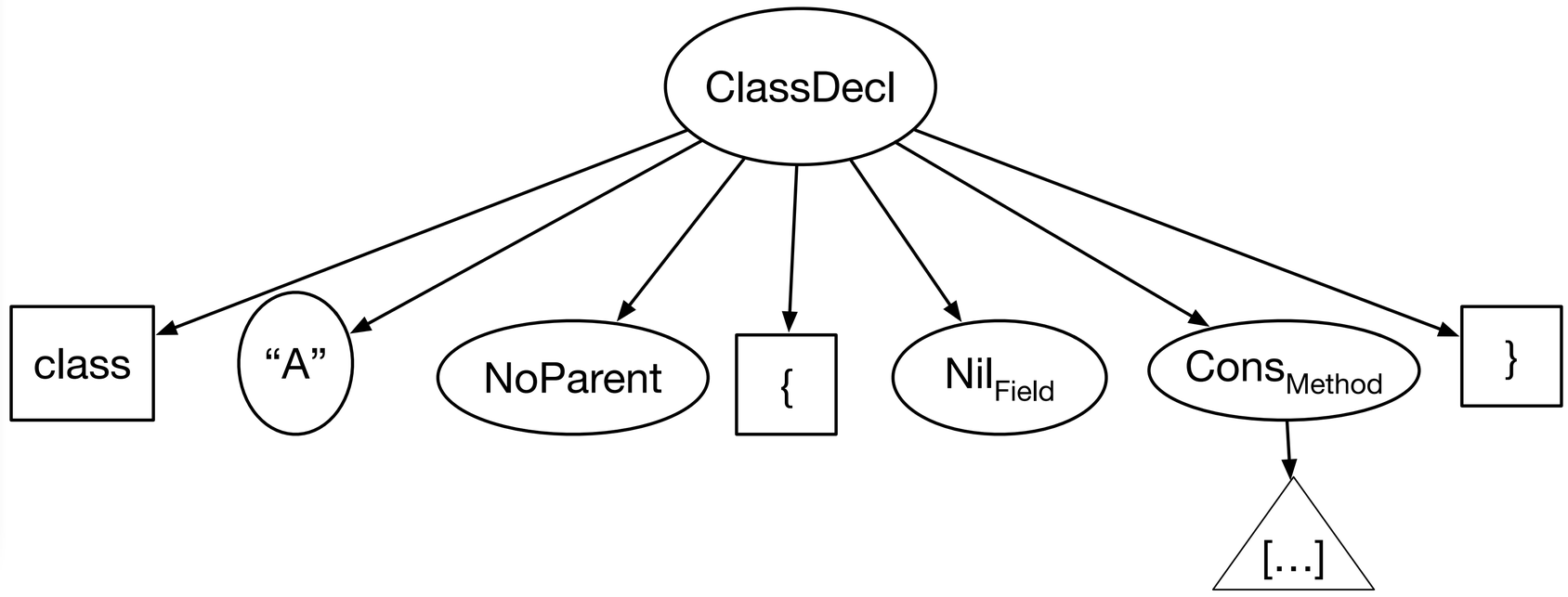
Inferring Placeholders

Placeholder Inference



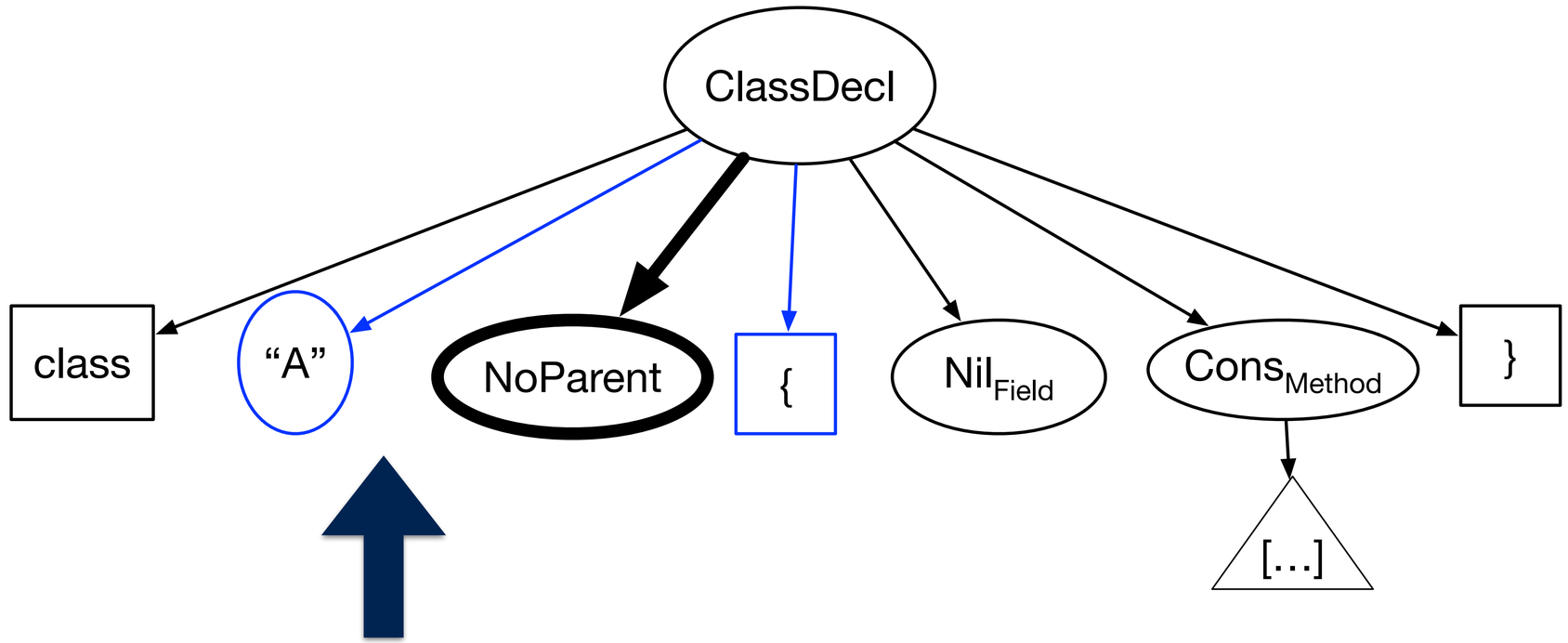
Placeholder Inference: Optional

```
class A {

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



Placeholder Inference: Optional

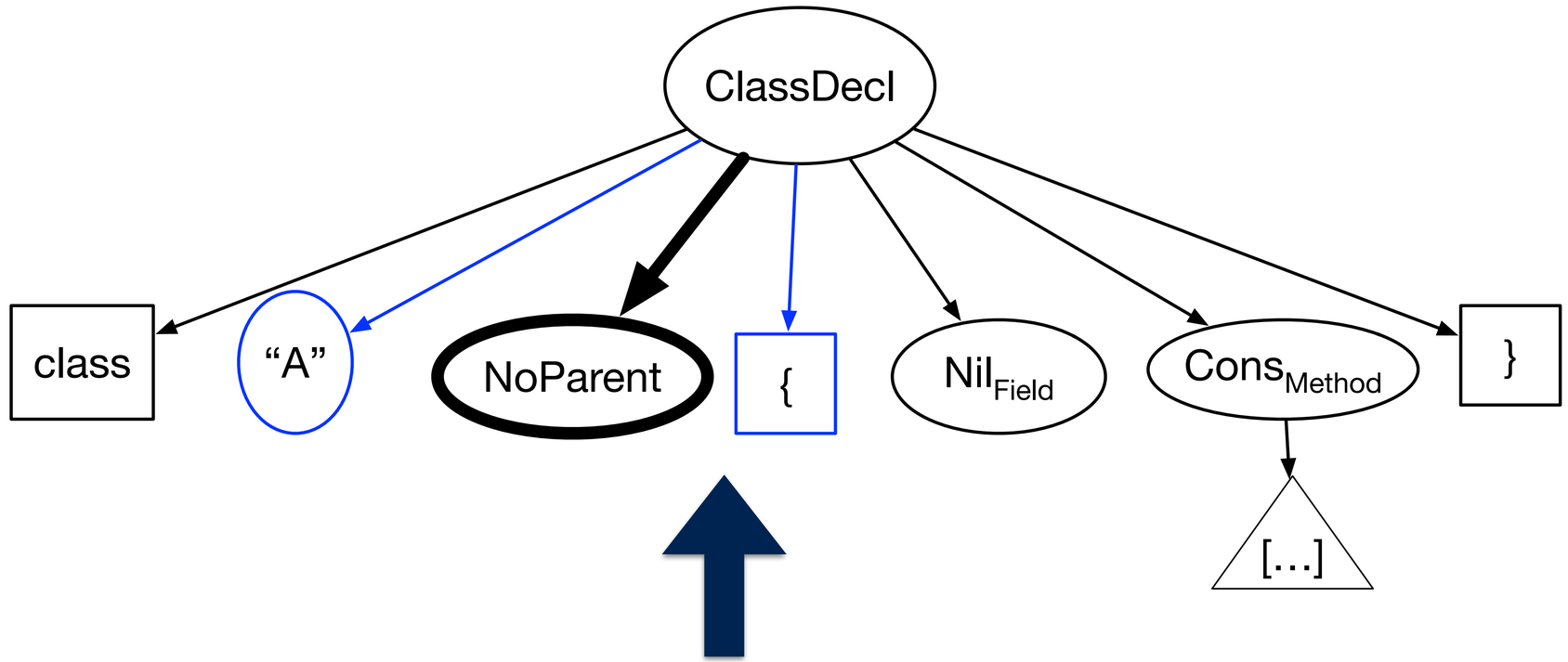
```
class A {

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



Placeholder Inference: Optional

```
class A {

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

```
class A {

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



Placeholder Inference: Optional

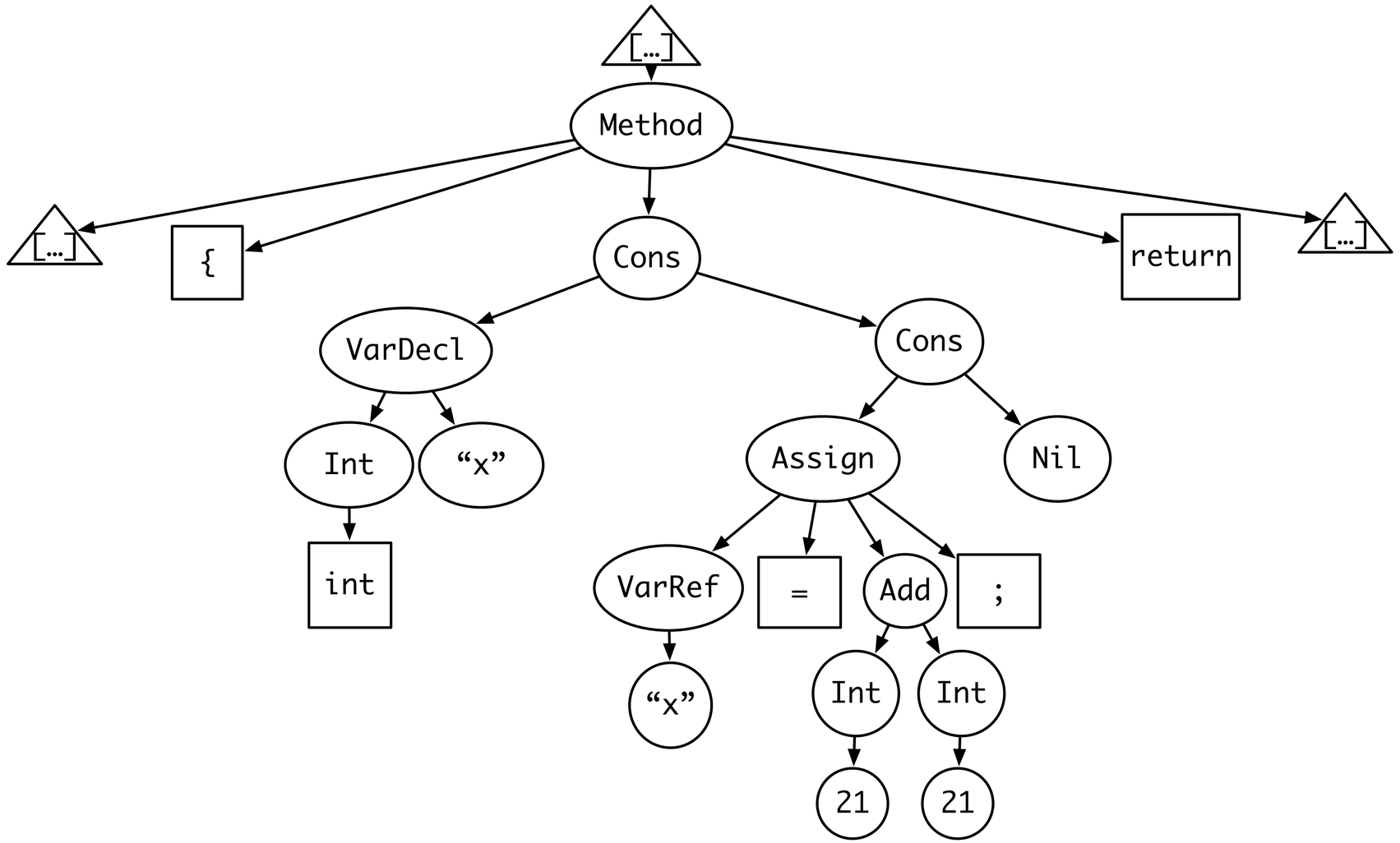
```
class A {
  public
    int
    x =
    retu
  }
}
```

```
Parent
extends $ID
```

```
class A extends $ID {

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

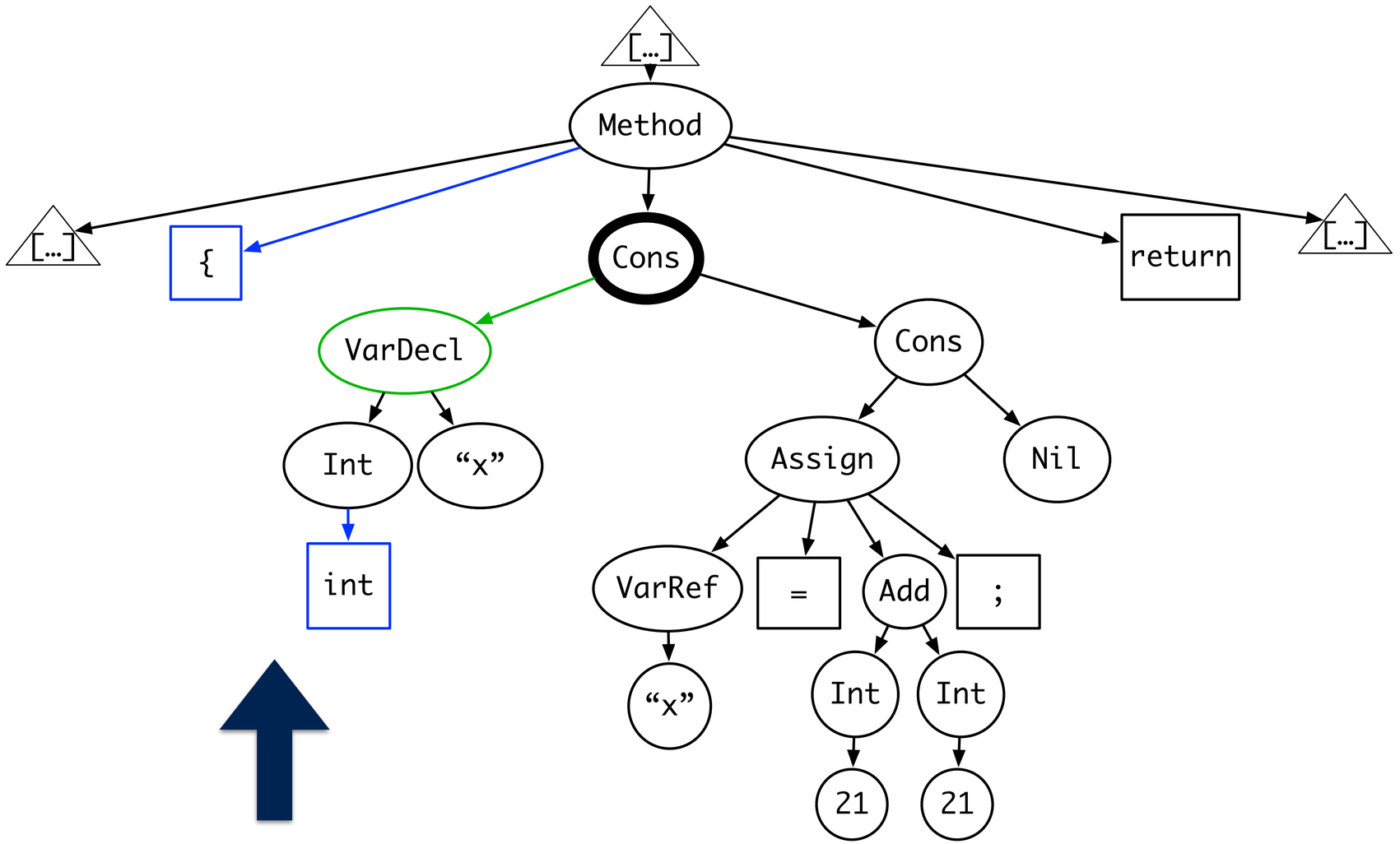
Placeholder Inference - Lists

```
class A {

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



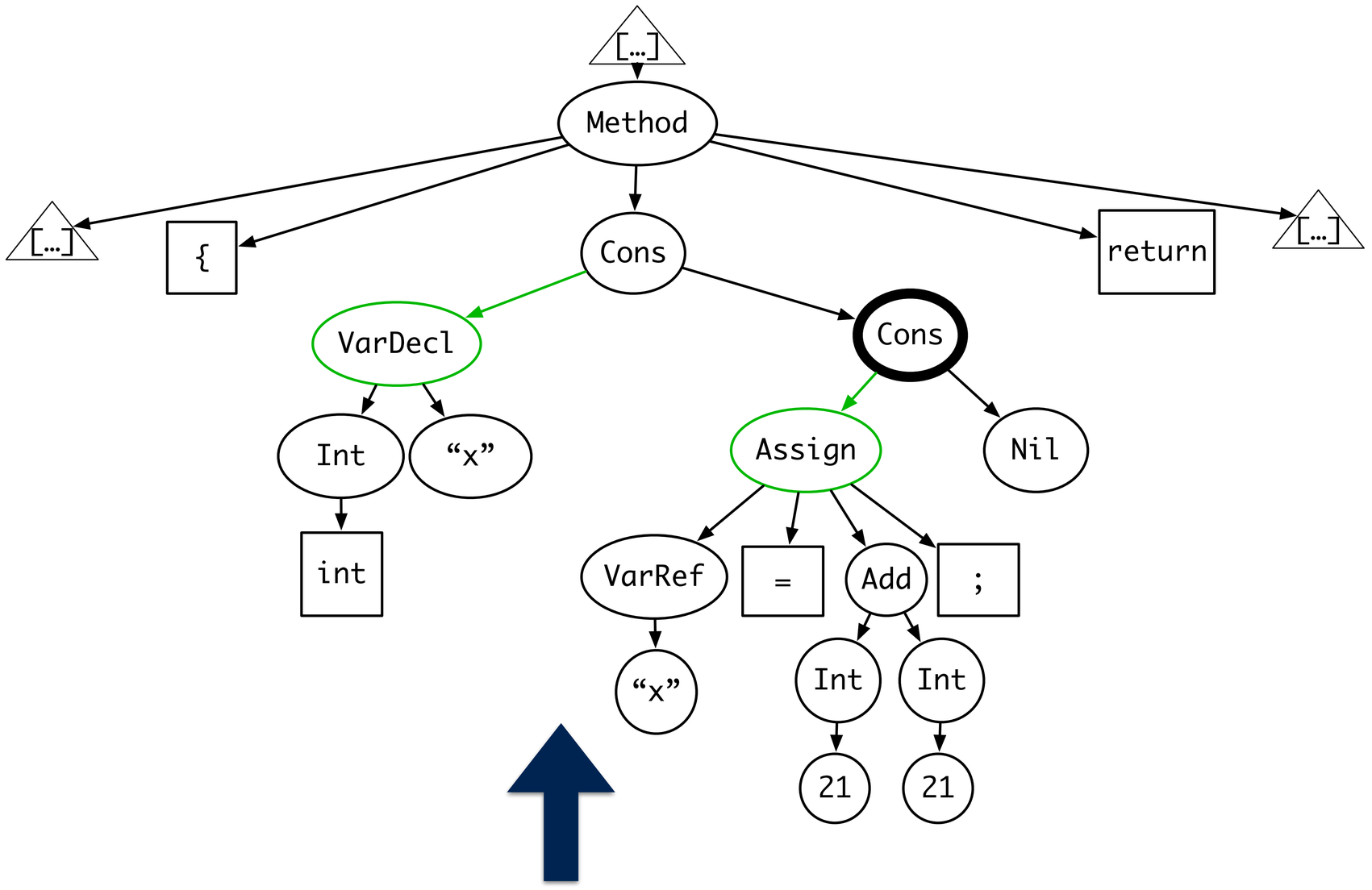
Placeholder Inference - Lists

```
class A {

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



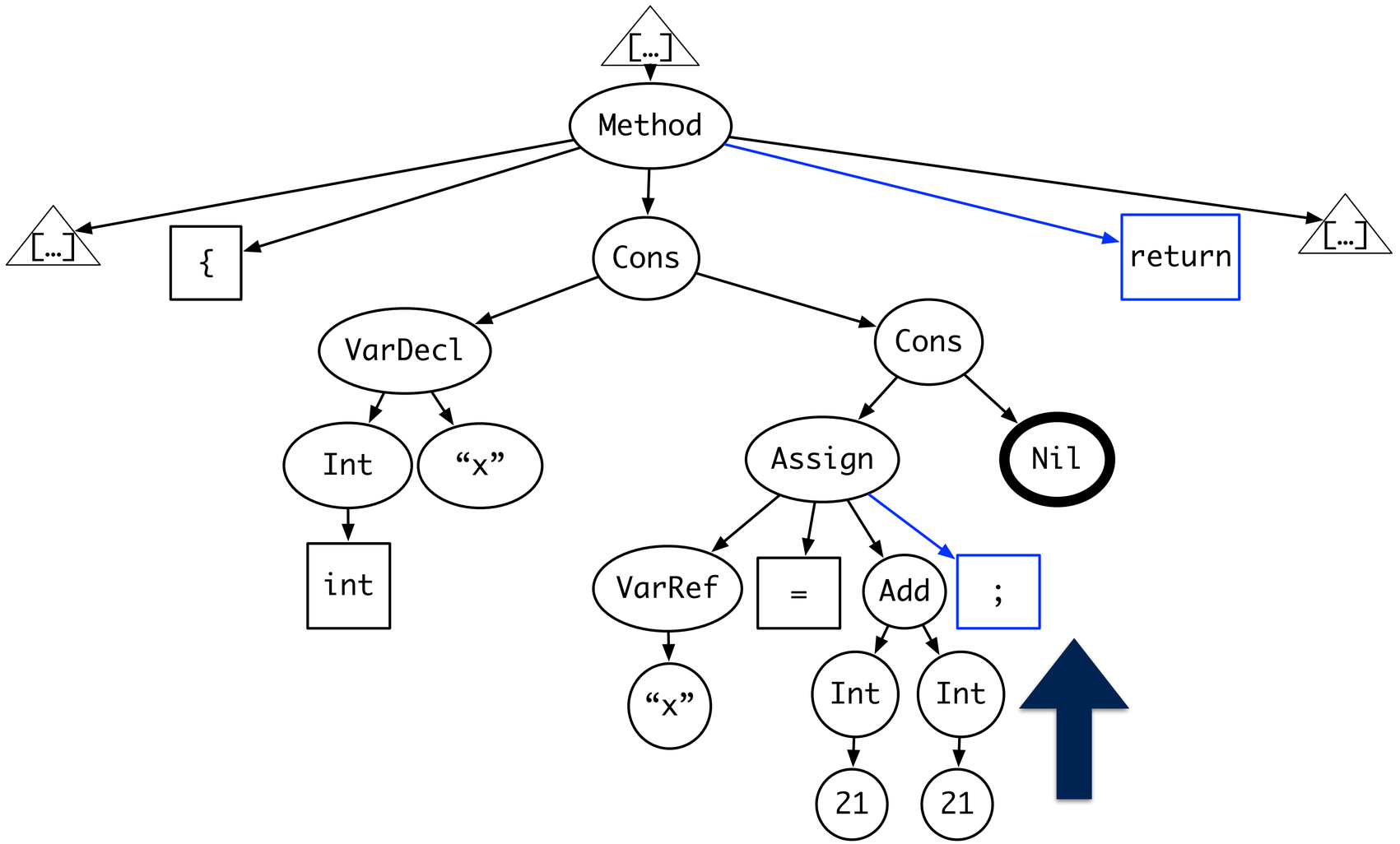
Placeholder Inference - Lists

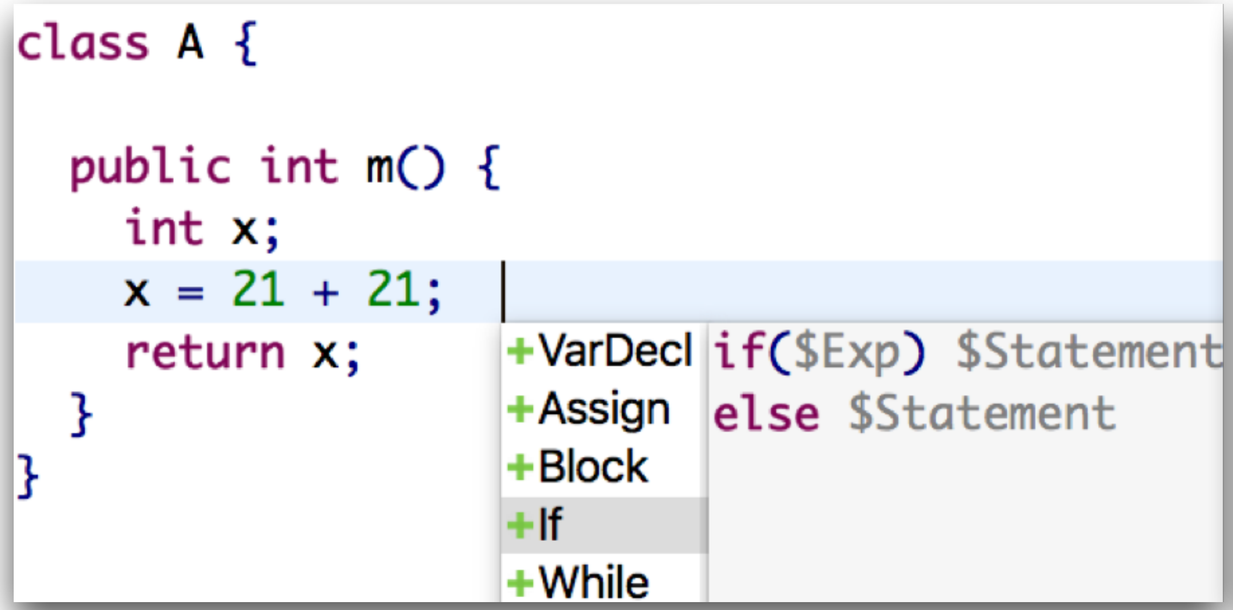
```
class A {

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

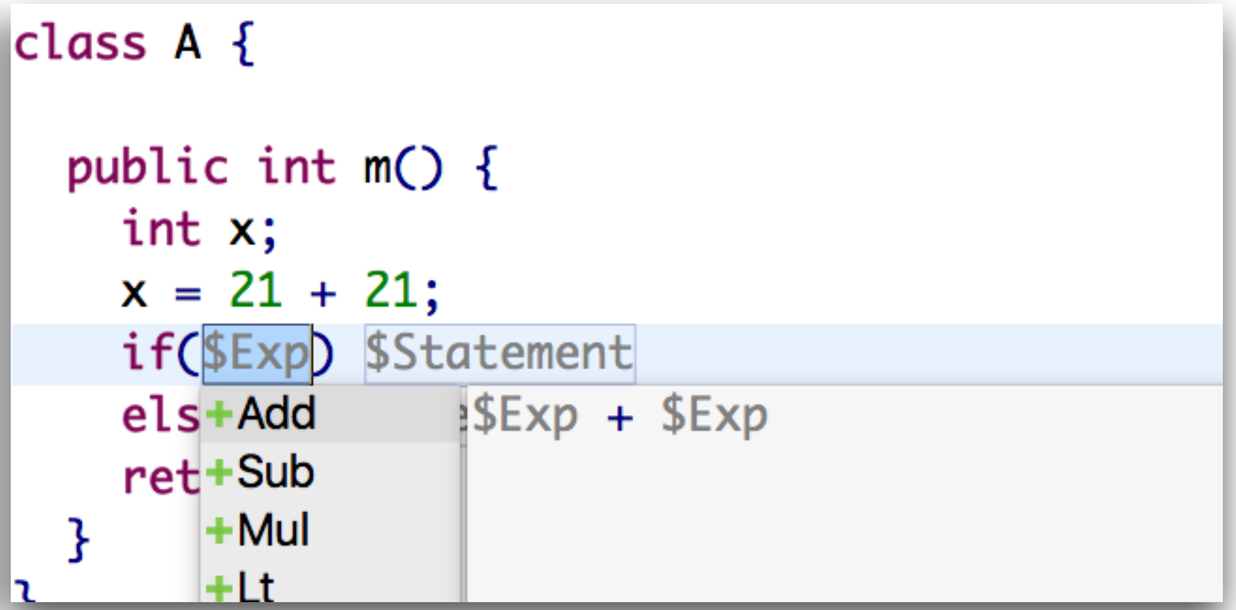


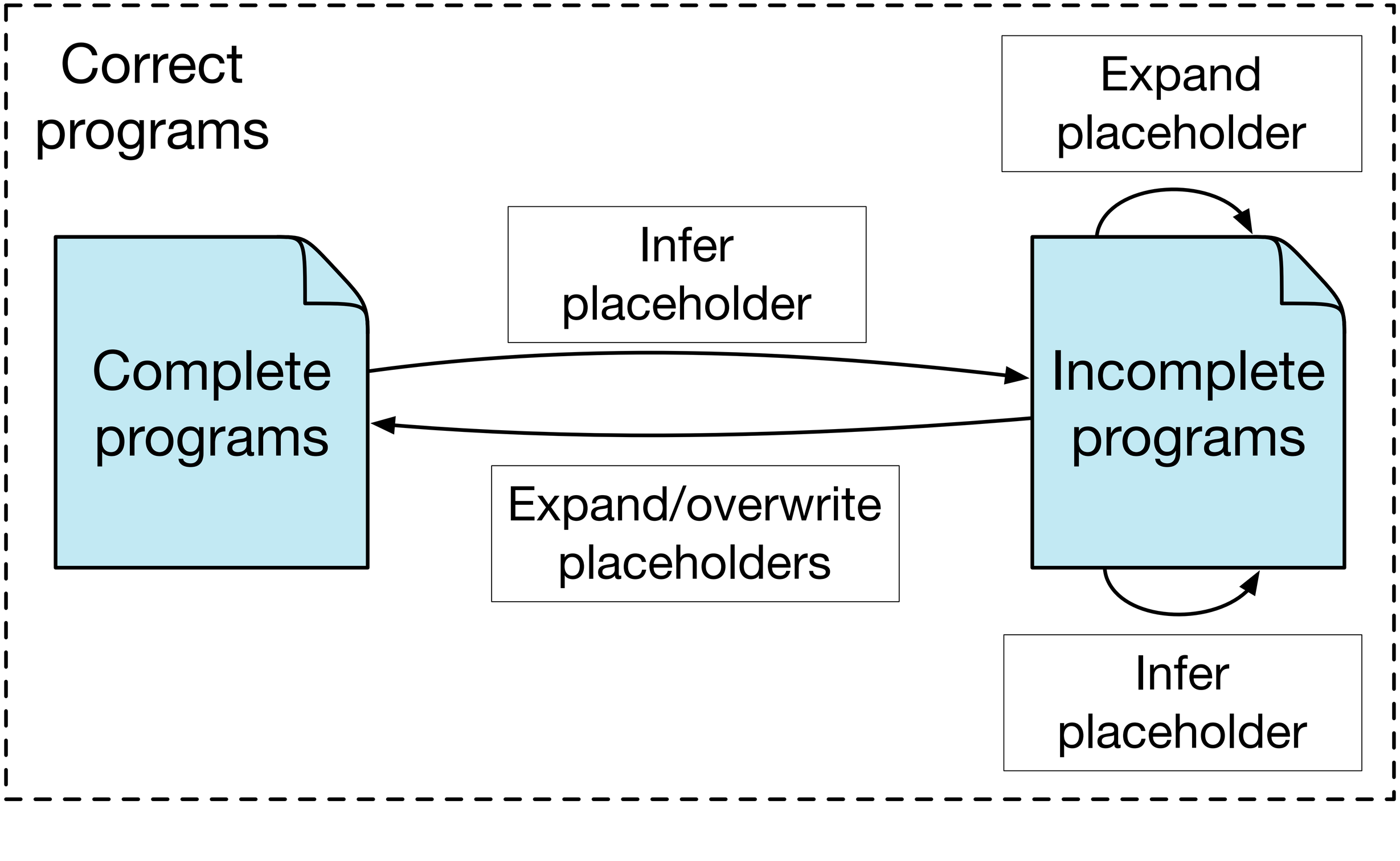
Placeholder Inference - Lists

```
class A {

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




class A {
public int m() {
int x;
x = 21 + 21;
return x;
}
}
+VarDecl
+Assign
+Block
+If
+While
if($Exp) $Statement
else $Statement


class A {
public int m() {
int x;
x = 21 + 21;
if($Exp) $Statement
+Add
+Sub
+Mul
+Lt
$Exp + $Exp


Correct programs
Expand placeholder
Infer placeholder
Complete programs
Incomplete programs
Expand/overwrite placeholders
Infer placeholder


```
class A {

  public int m() {
    int x;
      x | Syntax error, not expected here: ' x'
    return x;
  }
}
```

Idea 3

Error Recovery

```
class A {

  public int m() {
    int x;
    x |
    re
  }
}
```

- ✔ VarDecl
- ✔ Assign

```
x $ID;
```

```
class A {

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

```
class A {

  public int m() {
    int x;
    x |
    re
  }
}
```

- ✔ VarDecl
- ✔ Assign

```
x = $Exp;
```

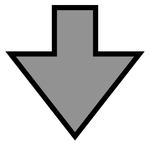
```
class A {

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

- + Add
- + Sub
- + Mul
- + Lt
- + VarRef

```
$Exp + $Exp
```

Insertion Rules

```
context-free syntax  //regular syntax rules

  Statement.VarDecl = <<Type> <ID>;>
  Statement.Assign  = <<VarRef> = <Exp>;>
```

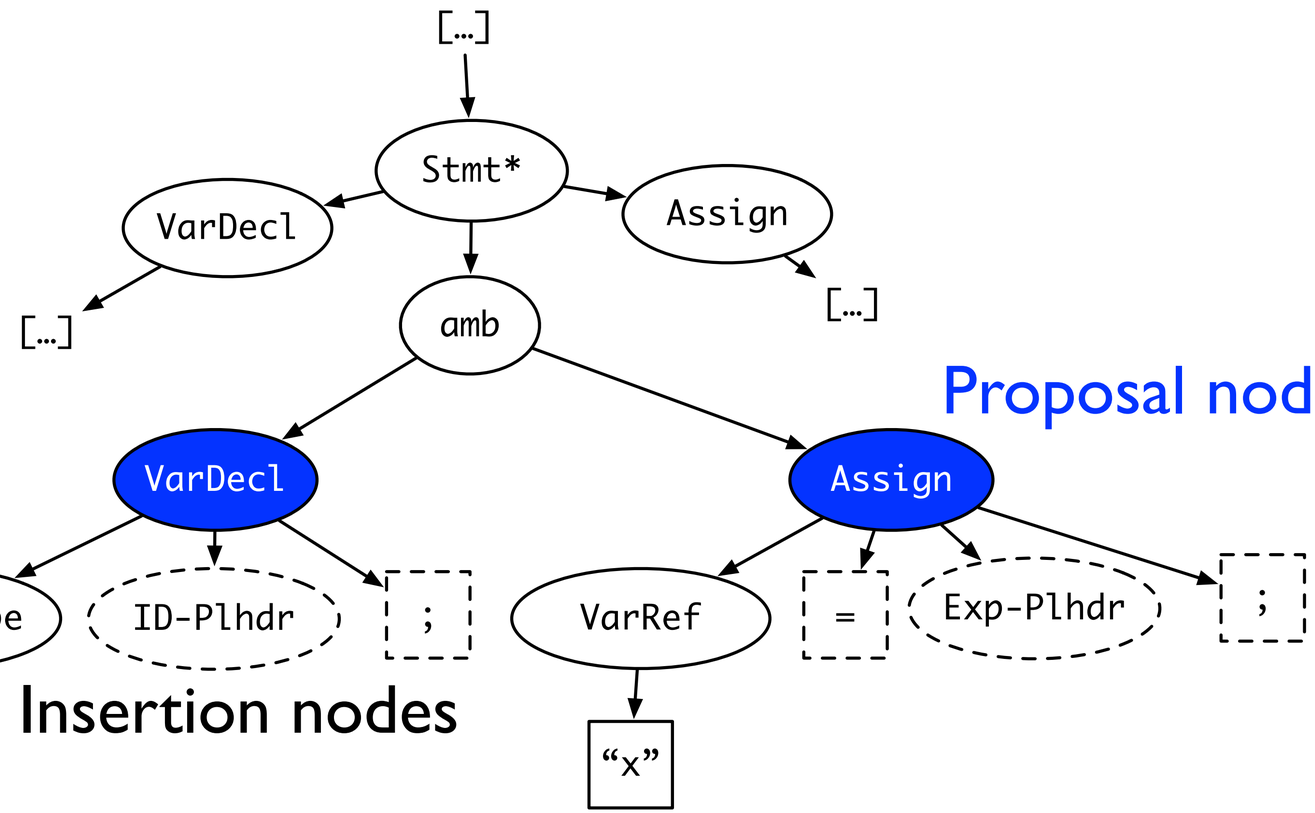


```
// derived insertion rules for placeholders
context-free syntax

  Type.Type-Plhdr           = {symbol-insertion}
  ID.ID-Plhdr               = {symbol-insertion}
  Statement.Statement-Plhdr = {symbol-insertion}
  VarRef.VarRef-Plhdr       = {symbol-insertion}
  Exp.Exp-Plhdr             = {symbol-insertion}

// derived insertion rules for literals
lexical syntax

  "=" = {symbol-completion}
  ";" = {symbol-completion}
```

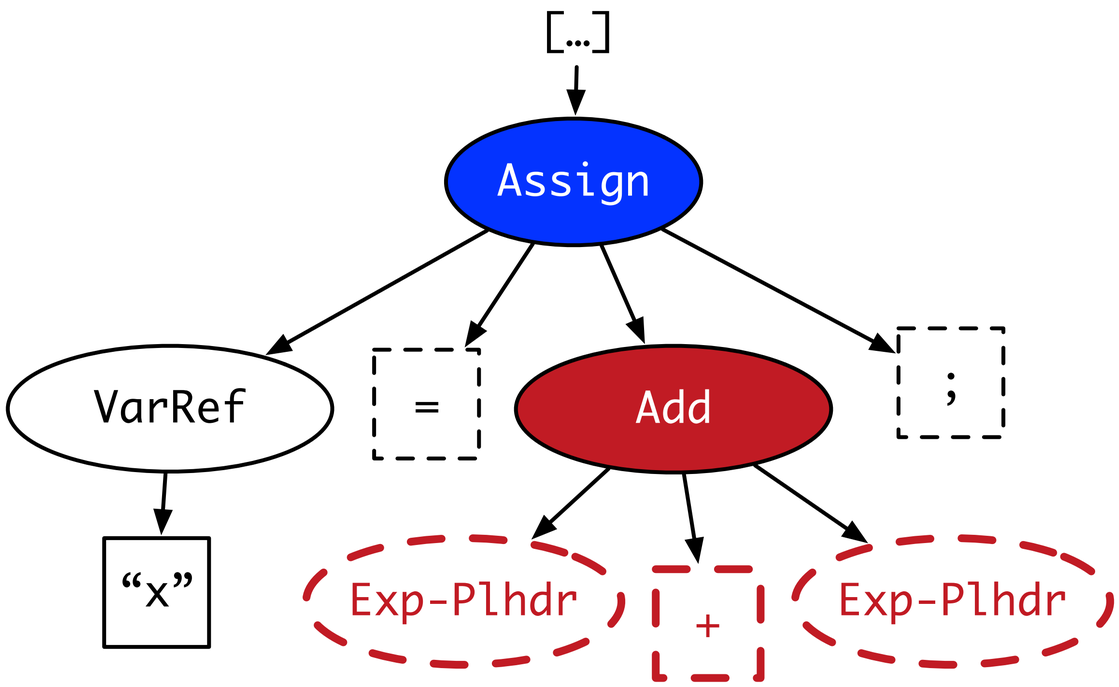
Empty productions

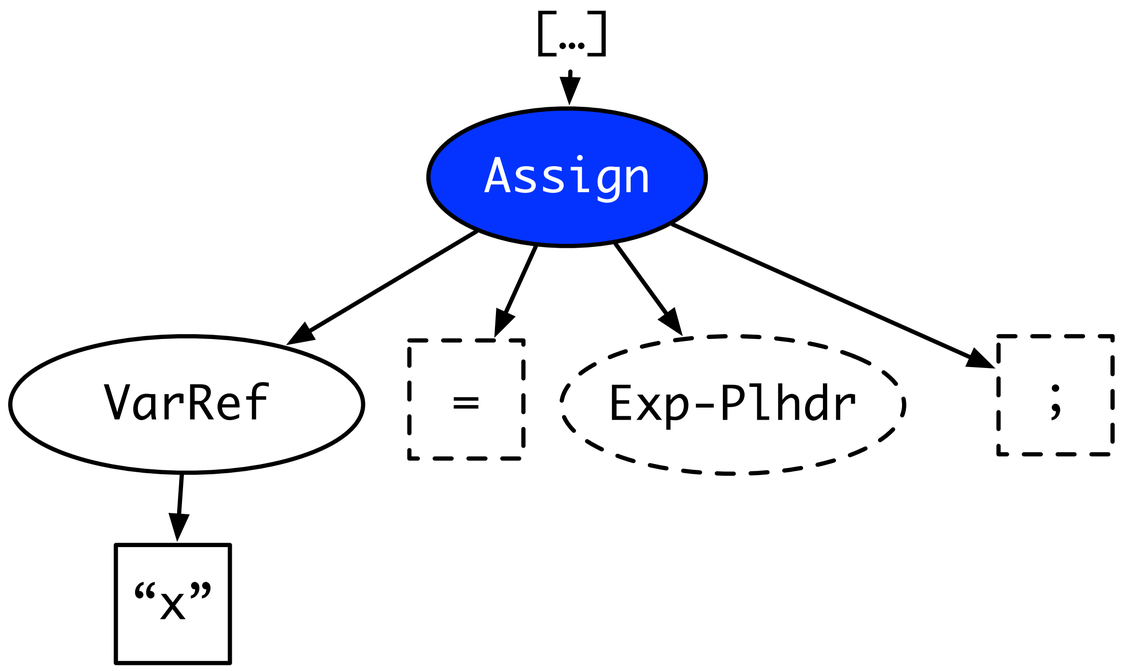
Apply Insertion Rules at Cursor

```
class A {

  public int m() {
    int x;
    x |
    re✔ VarDecl          x $ID;
    }  ✔ Assign
}
```



Limit Search Space

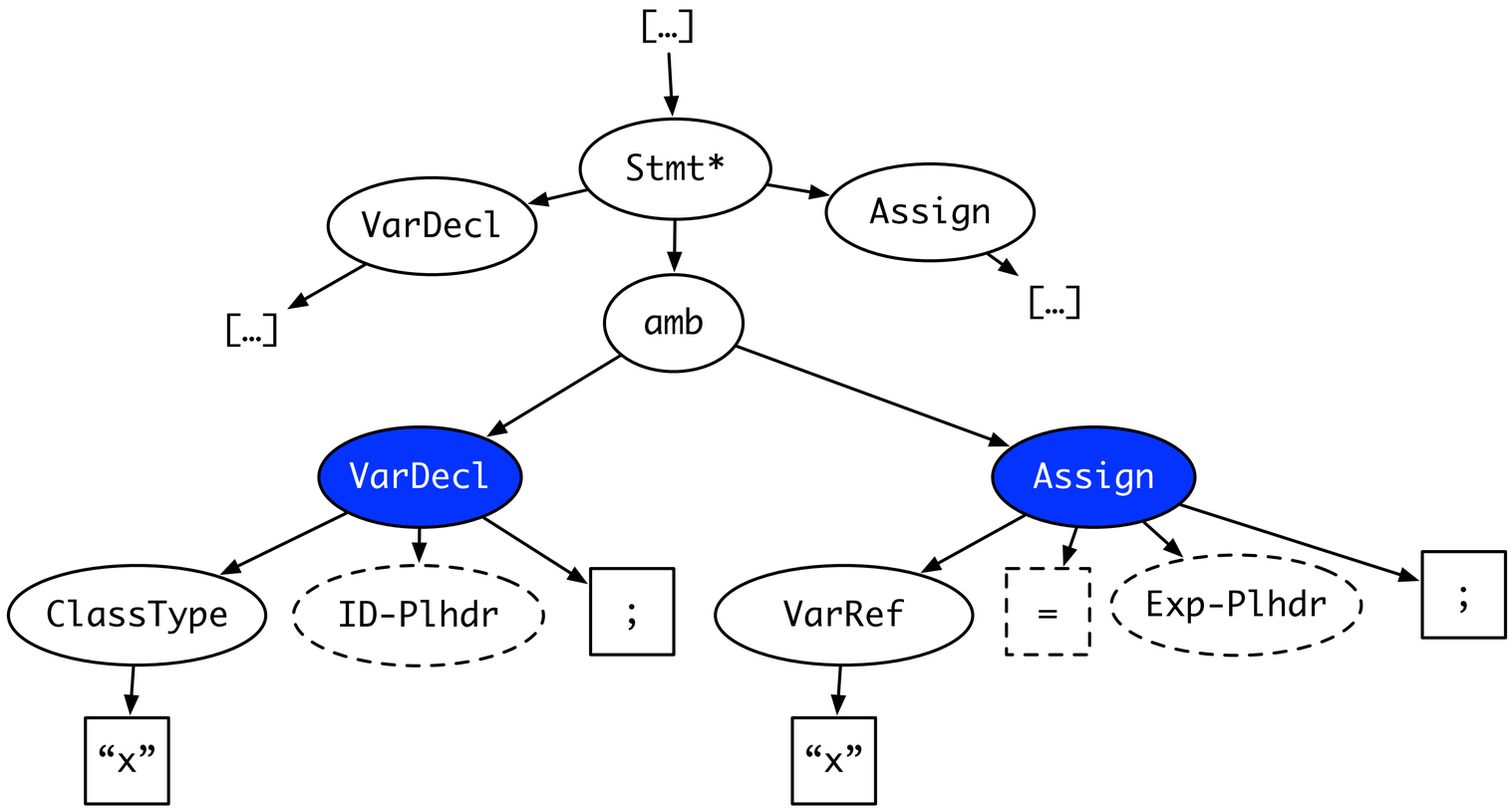




Use the simplest possible expansions

Greedy Recovery

```
class A {

  public int m() {
    int x;
    x ;
    re+ID-Plhdr                x = $Exp;
  }    ✔Assign
}
```



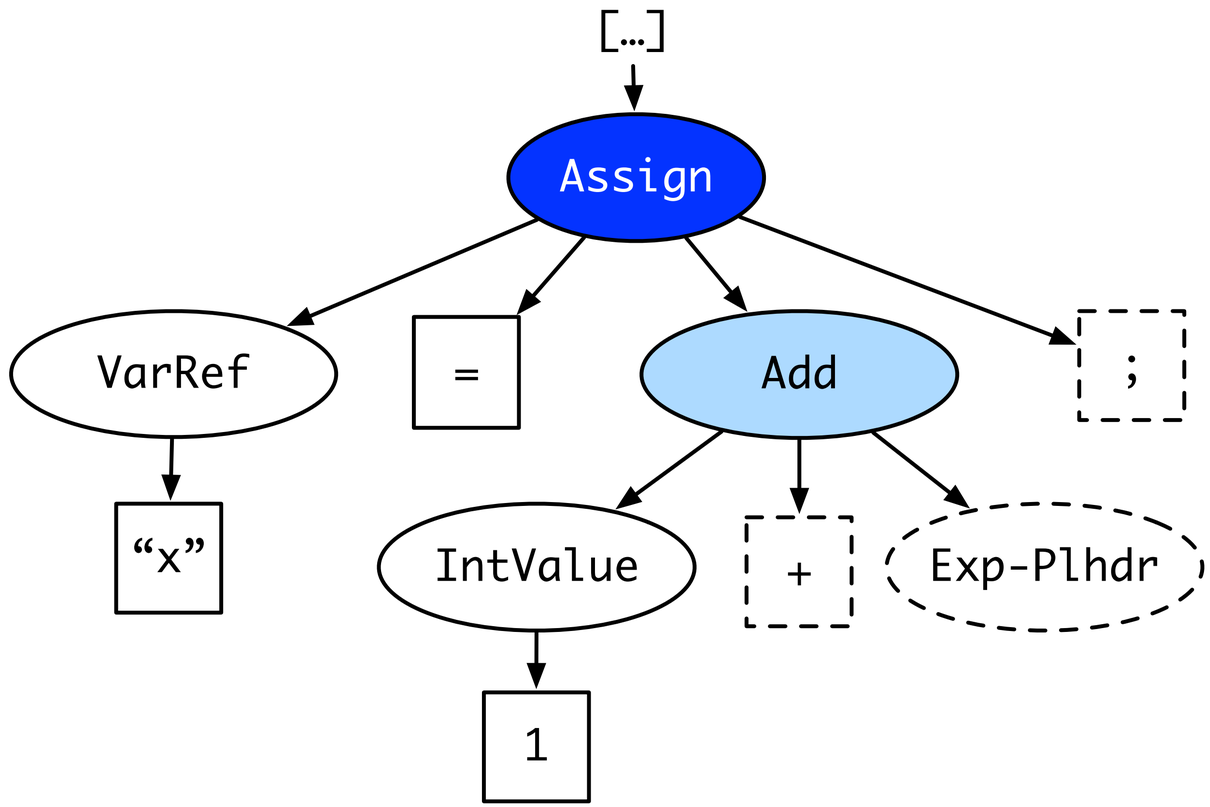
Include postfix in recovery proposal

Nested Proposal Nodes

```
class A {

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

Assign-Add

```
x = 21 + $Exp;
```

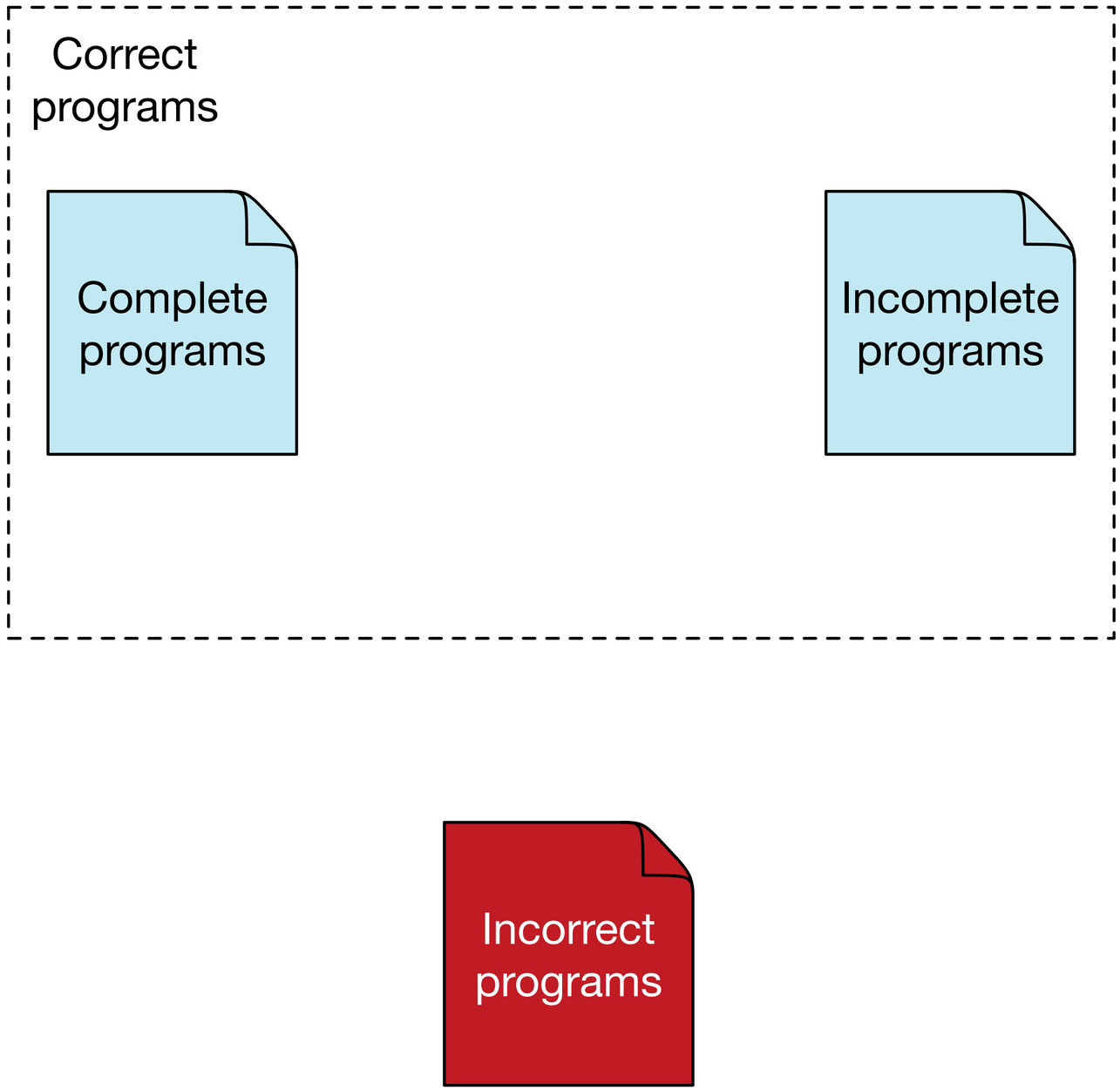


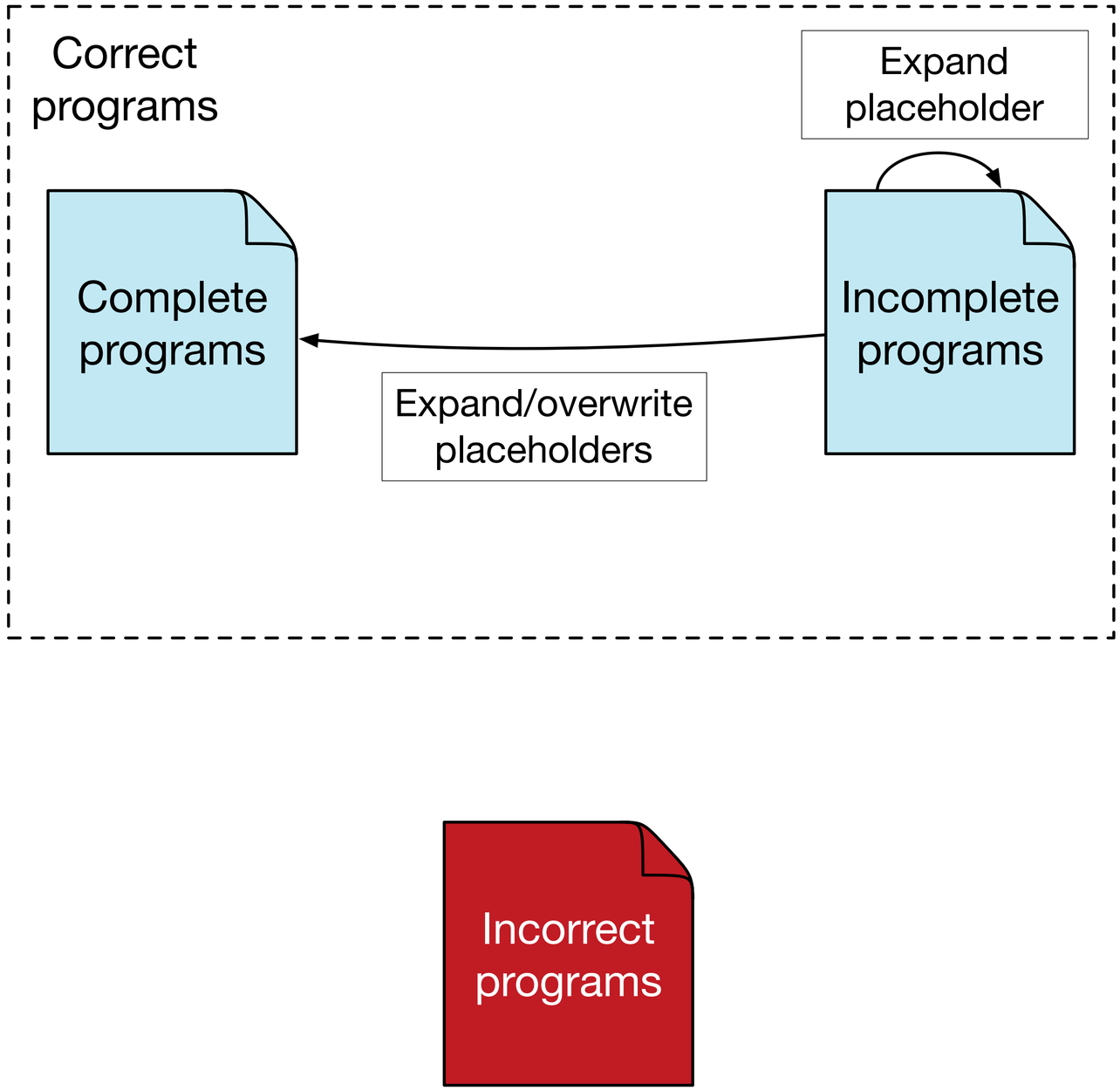
```
class A {

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

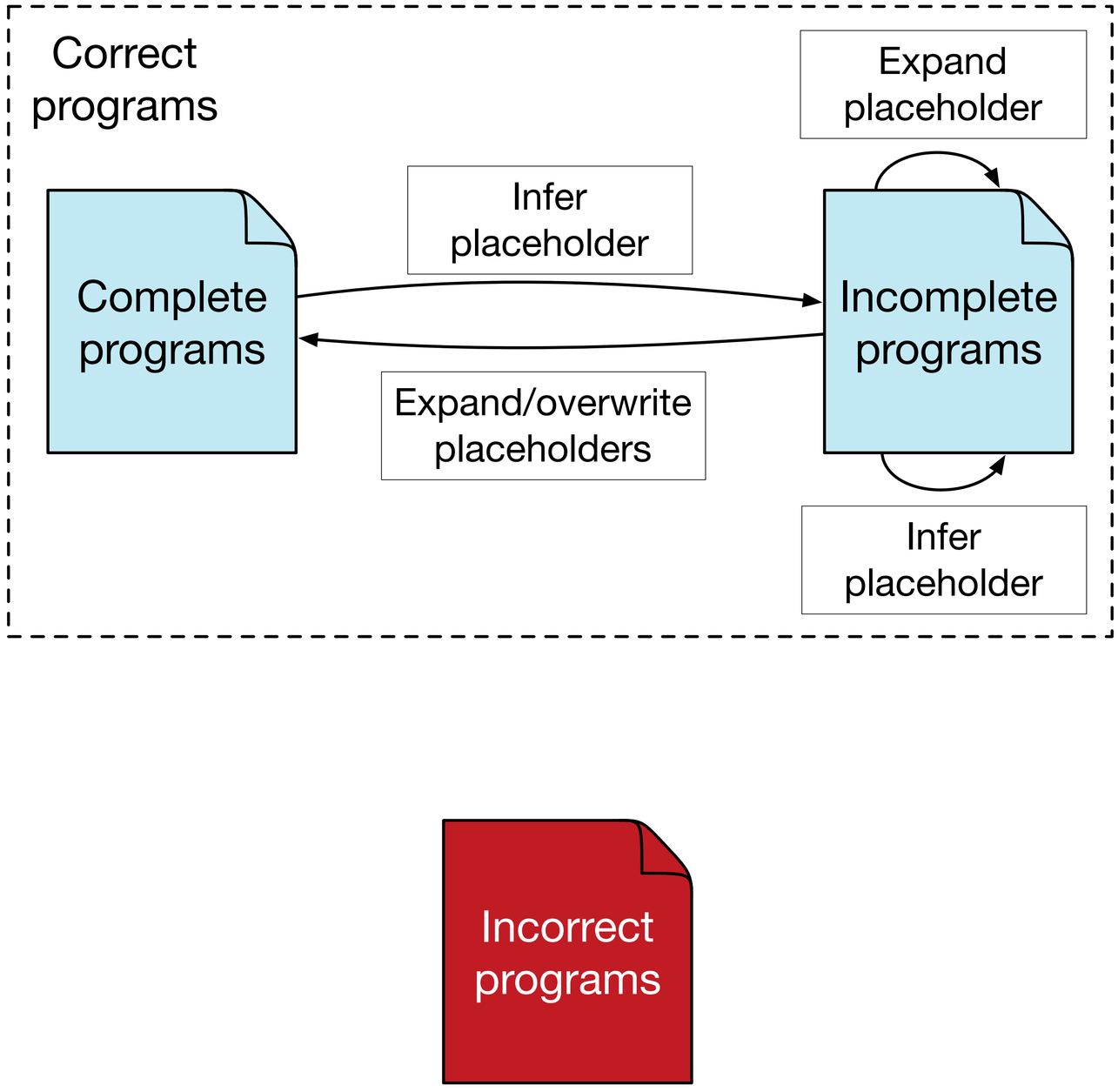
Assign-Add `x = 21 + $Exp;`

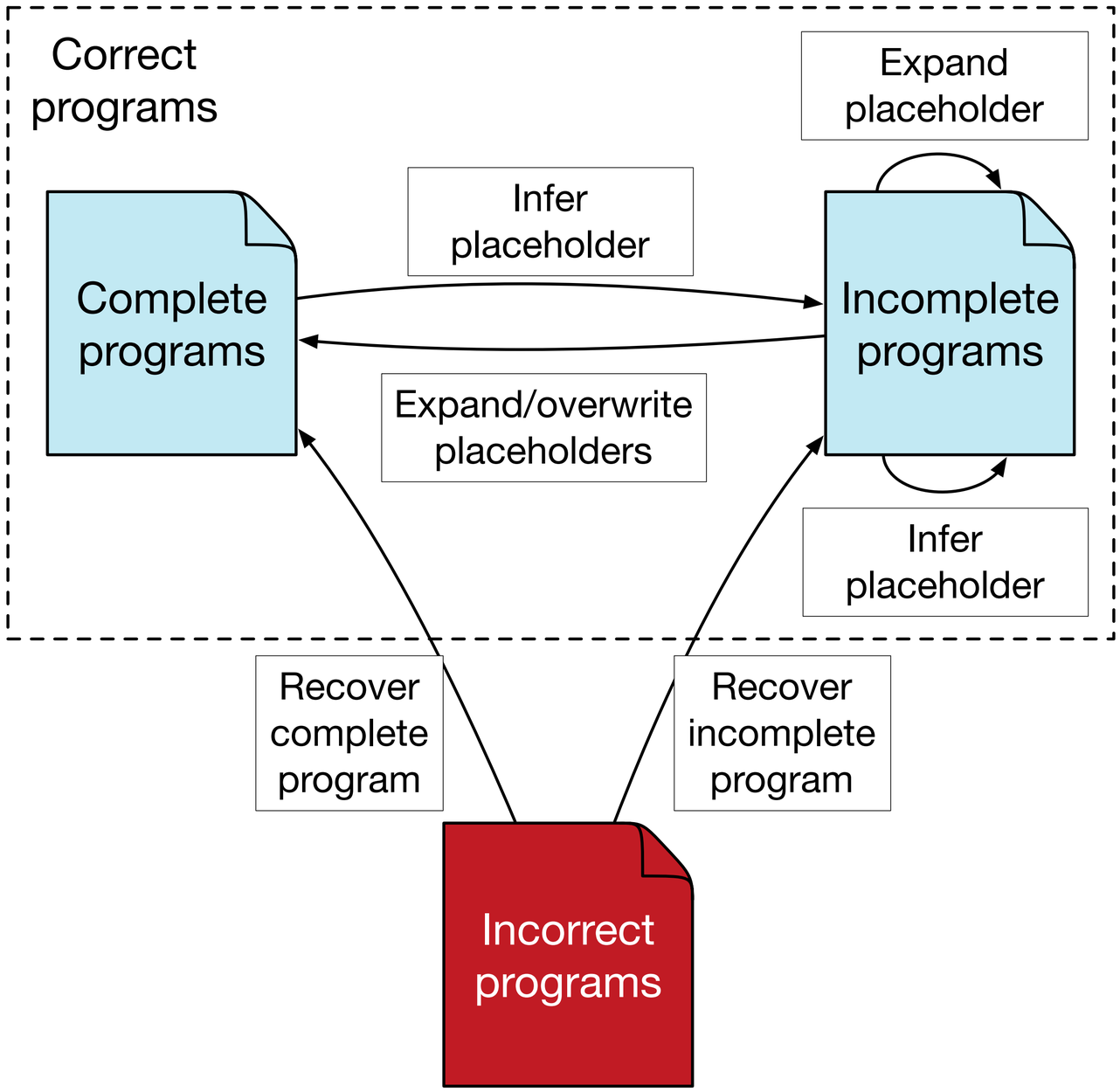
```
class A {

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

+Add `($Exp + $Exp)`
+Sub
+Mul
+Lt
+VarRef


Correct
programs
Complete
programs
Incomplete
programs
Incorrect
programs


Correct
programs
Expand
placeholder
Complete
programs
Incomplete
programs
Expand/overwrite
placeholders
Incorrect
programs


Correct programs
Expand placeholder
Infer placeholder
Complete programs
Incomplete programs
Expand/overwrite placeholders
Infer placeholder
Incorrect programs


Correct programs
Expand placeholder
Infer placeholder
Complete programs
Incomplete programs
Expand/overwrite placeholders
Infer placeholder
Recover complete program
Recover incomplete program
Incorrect programs

Syntax Definition: Summary

Separation of Concerns in Declarative Language Definition

Representation

- Standardized representation for <aspect> of programs
- Independent of specific object language

Specification Formalism

- Language-specific declarative rules
- Abstract from implementation concerns

Language-Independent Interpretation

- Formalism interpreted by language-independent algorithm
- Multiple interpretations for different purposes
- Reuse between implementations of different languages

Declarative Syntax Definition

Context-free grammars
- well-understood mathematical basis
- generation of valid sentences of language
- derivation corresponds to phrase structure

Character-level grammars
- terminals are characters
- lexical syntax defined using same (CFG) formalism

Declarative disambiguation
- associativity and priority declarations for phrase structure ambiguities
- follow restrictions and reject productions for lexical disambiguation
- not discussed: layout constraints for modeling layout sensitive languages

Structure
- abstract syntax tree schema defined in grammar using constructors

Formatting
- mapping from abstract syntax to text defined using template production

Services
- automatically derived: coloring, formatting, completion
- to do: parsing

Declarative Syntax Definition

Representation: (Abstract Syntax) Trees

- Standardized representation for structure of programs
- Basis for syntactic and semantic operations

Formalism: Syntax Definition

- Productions + Constructors + Templates + Disambiguation
- Language-specific rules: structure of each language construct

A meta-language for talking about syntax

Language-Independent Interpretation

- Well-formedness of abstract syntax trees
  - provides declarative correctness criterion for parsing
- Parsing algorithm
  - No need to understand parsing algorithm
  - Debugging in terms of representation
- Formatting based on layout hints in grammar
- Syntactic completion

Syntax of a Syntax Definition Formalism

Bootstrapping

Bootstrapping: SDF3 in SDF3

```
context-free syntax

  Grammar.Lexical = <
    lexical syntax

      <Production*>
    >

  Grammar.Contextfree = <
    context-free syntax

      <Production*>
    >

  Production.SdfProduction          = <<Symbol> = <Symbol*> <Attributes>>

  Production.SdfProductionWithCons = <<SortCons> = <Symbol*> <Attributes>>

  SortCons.SortCons = <<Symbol>.<Constructor>>
```

Exam Question

Exam Question

Consider the following SDF3 syntax definition and assume a standard definition for identifiers ID:

```
context-free syntax
  E.Var = ID
  E.Add = E "+" E
  E.App = E E
  E.Fun = "\\" ID "." "{" E "}"
  E     = "(" E ")" {bracket}
```

(a) Demonstrate that this grammar is ambiguous by giving two different abstract syntax trees using term notation based on the constructors in the grammar for the sentence f x + x and the left-most derivations that give rise to these trees.

(b) Disambiguate the syntax definition by means of priority and associativity declarations, i.e. without (otherwise) changing the productions. Motivate your choice of disambiguation rules. Is your disambiguation complete?

(c) Translate the disambiguated syntax definition into an unambiguous one without separate disambiguation rules. The syntax definition should accept the same language and produce the same abstract syntax trees.